UNIVERSITY OF OSLO
Department of Informatics

Vertically integrated models of CO_2 migration: GPU accelerated simulations

Martin Ertsås

May 6, 2011

Abstract

Storage of CO_2 in geological formations, such as oil and gas reservoirs, is considered an import means to reduce emissions of CO_2 into the atmosphere. Accurate modeling of the CO_2 migration is an important tool to analyse the risk of leakage in potential injection sites. To account for uncertainties in the geological model we need to run the simulations several times, with changes in the parameters, for the risk analysis. Along with several simulations we also need longer time scales than one normally has when simulation flow in porous medium, since the CO_2 should stay underground a lot longer than the time it takes to extract fossil fuels. Because of this, we do not only require an accurate model, but also a fast model for simulation. Using the vertical equilibrium assumption, and a vertically integrated model, has been shown to give good performance benefits with respect to the full 3D models used in the petroleum industry today, as well as being an accurate model.

In this thesis we will investigate how the GPU can be used as an accelerator unit for such vertically integrated models. We will compare the performance gained from using a GPU accelerated solver and a multi core solver, with respect to the performance from a serial solver. From this we will demonstrate that the GPU is a good accelerator unit for these model.

The solvers will be demonstrated to scale well both on simple grids with no faults, as well as on real world data with faults and geological traps for the CO_2. Lastly we will compare the error obtained on the GPU by using single precision floating point numbers instead of the double precision used on the CPU, and show that this error is negligible.

Preface

This master thesis has been written at SINTEF ICT as part of the MATLAB Reservoir Simulation Toolbox project [38]. It corresponds to approximately twelve months work over a period of one and a half year, with increasing focus on this thesis every half year. The work presented here is carried out individually, and builds upon the work done by Ligaarden and Nilsen [34] on vertically integrated models for CO_2 migration.

To work with such a project over a longer time scale has been a great experience, and has brought me many lessons, both personally and professionally, that will help me further in life. The thesis has resulted in long days and nights with little sleep, but I feel proud of the result presented in the end.

The computer codes developed have been tested to run on three different machines running different GNU/Linux distros (CentOS 5.5, Ubuntu 10.04 and Arch Linux) using gcc, as well as on machines running 32 and 64 bit Windows 7 compiled with Visual Studio 2005. Every test have been done using a MATLAB interface, called from MATLAB R2010a. As a consequence the computer codes are only verified to work using MATLAB R2010a, but should also work on newer MATLAB releases, and possibly some older.

To the reader: This thesis has been split into several chapters, which should be possible to read by themselves. However it is recommended to read the thesis sequentially, to get a grasp of the concept most efficiently. I have tried to give a thorough explanation as to why we use a vertically integrated model, but recommend readers to also read Ligaarden and Nilsen [34] to get a better understanding, as this thesis builds upon that paper.

Acknowledgements: First of all, I wish to thank my supervisors at SINTEF ICT, Knut-Andreas Lie and Jostein R. Natvig, who has provided me with guidance and advice whenever I needed it, I also appreciate all the red pens Knut-Andreas has sacrificed to make my thesis what it is today. My internal supervisor, Martin Reimers, also deserves to be thanked for always taking the time to sit down with me when needed. For providing me with a well written background paper, as well as help specific to CO_2 migration, I also have to thank Halvor Møll Nilsen and Ingeborg Ligaarden. This thesis would never have been possible in its current state without your help. A big thanks also goes to SINTEF ICT for providing me with a great working environment, both with regards to hardware and people, and the University of Oslo for holding far too many interesting courses to fit into five years, as well as providing me with

all the necessary licenses needed to write my thesis.

I wish to thank my friends for keeping me motivated, as well as helping me during my whole studies at UiO. I will especially thank my beloved Maren for keeping me sane during my most stressful periods, and making sure I also had a life on the side of my master thesis.

Lastly, I wish to thank my family for all the love, support and interest you have shown. A special thanks goes to my mom and dad who always have supported me in my decisions and helped me out when needed. I would not have been the person I am today had it not been for you.

Contents

Chapter 1

Introduction

In later years, public awareness on climate changes has increased, and several technologies have been proposed to help stop the, possibly human introduced, climate changes. One of these technologies is CO_2 Capture and Storage, which should help reduce CO_2 emissions to the atmosphere. For secure storage of CO_2 it is important with thorough risk analysis, to minimize the risk of leakage from the storage site, which means that simulations are important for the implementation of CO_2 storage.

In this thesis, we study the use of a GPU as an accelerator unit for simulating CO_2 migration in a porous medium. We will compare GPU accelerated simulations of a vertically integrated model to single- and multi-core CPU simulations.

1.1 Introduction

Today, we are releasing increasing amounts of CO_2 into the atmosphere, and while the CO_2 concentration has risen significantly since the Industrial Revolution, the temperature on earth has also risen slightly. There is reason to believe that these two phenomena are linked, and in 1992 the international concern about climate change led to the United Nations Framework Convention on Climate Change. The ultimate objective of this convention is the "stabilization of greenhouse gas concentration in the atmosphere at a level that prevents dangerous anthropogenic interference with the climate system" [47].

One of the techniques considered to help reduce emissions to the atmosphere is CO_2 Capture and Storage (CCS). This technique consists of capturing CO_2 during the combustion and extraction processes, transporting the captured CO_2 to an injection site, and injecting it into the subsurface. The UN have concluded that CCS is an important step to reduce CO_2 emissions, as well as being an economically feasible solution [47]. They also point out that there are security risks one have to take into consideration, like leakages from the injection site and cracks made from the increase in pressure. This calls for thorough risk analysis, where computer simulations play a huge role. The focus of this thesis

will be on CO_2 storage, and more specifically simulating the migration of the injected CO_2.

CO_2 storage means, for our problem, injecting CO_2 into a geological rock formation, like oil and gas reservoirs or saline aquifers. The technology required is the same as used when injecting water or gas for enhanced oil and gas recovery, which means that the problem is already well understood and the technology present [4]. CO_2 storage has also been deployed in large-scale commercial applications, as well as smaller research projects. To date, there exists four large facilities for CO_2 storage: Weyburn-Midale, Sleipner, Snøhvit, and In Salah, which combined inject $\sim$6.4 million tonnes of CO_2 every year [8, 13, 46, 49]. Another formation considered for CO_2 storage is the Johansen formation outside of Norway, which can sustain a yearly injection of $\sim$3 million tonnes of CO_2 [2]. Clearly, the amount of CO_2 vented into the atmosphere will be greatly reduced by implementing CO_2 storage in large scales. One example of the impact obtained by CCS is an onshore field outside of Longyearbyen on Svalbard. This field is a key part in the goal to make Longyearbyen the world's first CO_2 neutral city by 2025, where CO_2 will be captured from the coal-fueled power plant and injected into this formation [48, 50].

Alternatives to injecting CO_2 into rock formations as described, is to inject CO_2 into coal seams and into the ocean at depths greater than 1,000m. Both of these techniques are still in a research phase, and will not be discussed further in this thesis [4, 9].

Reservoir simulations have been a part of oil-reservoir management for more than fifty years [1], and these kind of simulations can be used to simulate migration of CO_2 in oil and gas reservoirs. The biggest difference between the reservoir simulations done in oil and gas recovery, and simulating CO_2 migration, is the different time scales. When simulating CO_2 migration it is important to make sure there is no leakages, and one have to simulate until almost all the CO_2 is trapped, which might take several thousand years. When simulating for oil and gas recovery on the other hand, one seldom simulates for more than one century, This means we need both fast and reliable simulations for CO_2 migration problems.

Models based on the vertical equilibrium (VE) assumption have long traditions for describing flow in porous media, and was early on extended to handle two-phase and three-phase flow [16, 36, 37]. As computational power has increased, models based on the VE assumption have been less used, and one have instead used the full 3D model. Recently, there have been a renewed interest in VE based models for fast simulation of CO_2 migration, when an assumption of a sharp interface between CO_2 and brine may be reasonable [14, 22, 23, 34, 35].

The usage of graphics processing units (GPUs) as accelerator units for hyperbolic partial differential equations, for which explicit schemes are feasible, have proven to give good performance benefits over a CPU only implementation [6, 10, 12]. There has also been done some work on GPU accelerating implicit transport, which have proven to give good performance benefits both with respect to computing the preconditioner and solving the arising linear system [5, 31], but not as good as for the explicit case. It is reason to believe that the equation that arises in our problem, will be well suited for explicit transport, as well as

parallel computing and GPU acceleration.

Further in this thesis, we will use the vertical equilibrium assumption to make a fast simulation model. The arising model will be discretized and implemented on the CPU for both serial and parallel execution, together with a GPU accelerated implementation. The main goal is to see how suited this problem is for GPU acceleration, and how the usage of single-precision floating-point numbers affect the results. The thesis is based on a paper by Ligaarden and Nilsen [34], where they explored the numerical aspects of using the vertical equilibrium assumption for simulating CO_2 sequestration. They believe that making GPU accelerated solvers for this problem will give the possibility of large-scale desktop simulations of VE models on coarse grids.

1.2 Precision and accuracy

The words *precision* and *accuracy* will be key words in parts of this thesis, we will therefore state their definition here. In the domain of computing, Websters dictionary defines:

Accuracy: *How close to the real value a measurement is.*

Precision: *The number of decimal places to which a number is computed.*

In this thesis we will tacitly assume that the double precision value is the real answer, and use the word accuracy to describe how close to the double precision answer our single precision solution is. The fixed precision on a computer means that we only have a finite set of numbers to represent the infinite set of real numbers. Further this means that the lower the precision is, the more limited the finite set is, and so the set made up of the single precision floating point numbers is a lot smaller than the set made up of the double precision floating point numbers. Clearly the accuracy is highly dependent on the precision, and we do not expect our single precision answer to be the same as the double precision answer.

1.2.1 Unit of least precision

Unit of least precision (ULP), is a measure of precision in numerical calculations. Its definition was originally defined as:

ULP [29]: *ULP(x) is the gap between the two floating-point numbers nearest x, even if x is one of them.*

By the introduction of the IEEE 754 floating-point standard, the definition of ULP had to change to handle NaN and Inf. The new definition was:

ULP [29]: *ULP(x) is the gap between the two finite floating-point numbers nearest x, even if x is one of them. (But ULP(NaN) is NaN.)*

The IEEE 754 floating-point standard requires that the results of an elementary arithmetic operation (addition, subtraction, multiplication, division, and square roots) should be withing 0.5 ULP of the mathematically exact answer, meaning

we will round to the nearest floating-point in the given precision.

Each time we do a single precision arithmetic operation, we will gain a round-off error with respect to the double precision, due to the ULP. The error will be small for each operation, but as the number of operations increase this error will be accumulated, and thereby grow. This means we might end up with a large error, even though the error introduced for each operation is small.

1.3 Research questions

The goal of this thesis is to give complete and precise answers to the following questions. The questions will be summed up in Chapter 5.

i. *How good can a simulation using the vertical equilibrium assumption utilize the GPU hardware?*

ii. *How does the performance scale on the GPU with respect to the horizontal and vertical resolution?*

iii. *How does the performance scale on the GPU with respect to the inclusion of faults as well as varying geometry in the reservoir?*

iv. *Are the results obtained using single precision on the GPU accurate enough for practical purposes?*

The first question will be answered by looking at the occupancy of the GPU during execution. The second and third question will be answered by running several test-cases using both idealized grids and grids with realistic features, and compared to both a serial and a multi-core CPU solution. The last question will be answered by comparing the single-precision GPU solution to a double-precision multi-core solution, using a grid of the Johansen formation outside of Norway.

The thesis is partitioned into five chapters:

Chapter 1: A short introduction to the thesis.

Chapter 2: Gives the background needed for the work done in this thesis. This includes more on CCS, reservoir simulation, the vertical equilibrium assumption, and GPU programming.

Chapter 3: Shows how the standard transport and pressure equations can be vertically integrated, and uses the finite volume method to get an explicit scheme. Also discusses alternative algorithms, as well as GPU specific problems.

Chapter 4: Reports and dIscusses the test results from our numerical tests.

Chapter 5: Concludes the findings from our tests, and discusses further work on this subject.

1.4 Notation

The notation used throughout the thesis, will mostly be standard notation for PDEs. To prevent confusion however, we will still establish it formally.

Our domain will be denoted Ω, and the size of the domain will be denoted $|\Omega|$. Further, a discrete cell in a 2D domain will be denoted $C_{i,j}$, where $C_{i,j} = [x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}]$ will be the ith cell in the x-direction and the jth cell in the y-direction. We also have the discrete sizes $\Delta x_i = x_{i+1/2} - x_{i-1/2}$ and $\Delta y_j = y_{j+1/2} - y_{j-1/2}$, which is the size of a cell in x and y directions, respectively, as well as the size of a cell $|C_{i,j}| = \Delta x_i \Delta y_j$. In our discrete problem, we are only interested in two dimensions, here is therefore no need to establish an equivalent notation for a 3D space.

Along with the space domain, the time domain is also discretized. We let $t_n = \sum_{l=0}^{n-1} \Delta t_l$, where $\Delta t_l = t_{l+1} - t_l$.

In this thesis, x and y will mean the normal x and y directions in a 2D coordinate system while $\mathbf{x}$, that is a boldface x, will mean the directional vector (x, y). This will also be used further in integration, where:

$$\int_{C_{i,j}} f \, d\mathbf{x} = \int_{x_{i-1/2}}^{x_{i+1/2}} \int_{y_{j-1/2}}^{y_{j+1/2}} f \, dx \, dy$$

We also have to establish some notation for derivatives. In 3D we use the normal notation $\frac{\partial f}{\partial z_i}$ as the derivative of a function f in direction z_i, where z_i can be either time t or a spatial coordinate x,y,z. Further we have $\nabla f = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z})$. The divergence will be denoted $\nabla \cdot f = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y} + \frac{\partial f}{\partial z}$. We will use a vertical equilibrium assumption on our 3D model to get a 2D model, during this process we will introduce the derivatives $\nabla_{||} f$ and $\nabla_{||} \cdot f$. These derivatives is the same as for 3D, but in two dimensions, and the directions are now parallel to a surface, in this case the top surface of the reservoir.

Chapter 2

Background

Here we will give the necessary background for the rest of the thesis. We will start by a short introduction to CCS, before giving a background in reservoir simulation, the vertical equilibrium assumption and General-Purpose computation on Graphics Processing Units (GPGPU).

2.1 CO_2 Capture and Storage

CO_2 Capture and Storage consists of three parts, capture of CO_2, transportation of CO_2, and injection of CO_2.

The capture of CO_2 is likely to be applied to large point sources like fossil fuel plants, fuel processing plants, and large industrial plants. Capturing CO_2 from small and mobile sources, such as in transportation and the building industry, is likely to be more difficult to implement, as well as too expensive with regards to the contribution this will give to the environment. There are four basic systems for capturing CO_2 from uses of fossil fuels and biomass: Capture from industrial process streams, post-combustion capture, pre-combustion capture and oxy-fuel capture [3].

CO_2 can be transported in either gas, liquid, or solid state. The most common transportation alternative is to use pipelines or ships for transport of gas and liquid CO_2. In atmospheric pressure, CO_2 takes up a large volume, which means that large facilities are needed. To reduce transport costs, one can increase the pressure, which means that more CO_2 can be transported in the same volume. To further decrease the volume needed one can liquefy the CO_2, which is needed if ships are used for transportation. Liquefaction is an established technology for natural and petroleum gases, and the existing technology can be used to liquefy CO_2. Solidification of CO_2 is currently too expensive to be feasible [17].

The differences in transportation cost using ships and pipelines can be seen in Figure 2.1. One can clearly see that using pipelines is the most economical solution if the distance is less than $\sim$1,000 km, which means that pipeline transportation is the most economical for the CO_2 storage facilities in use.

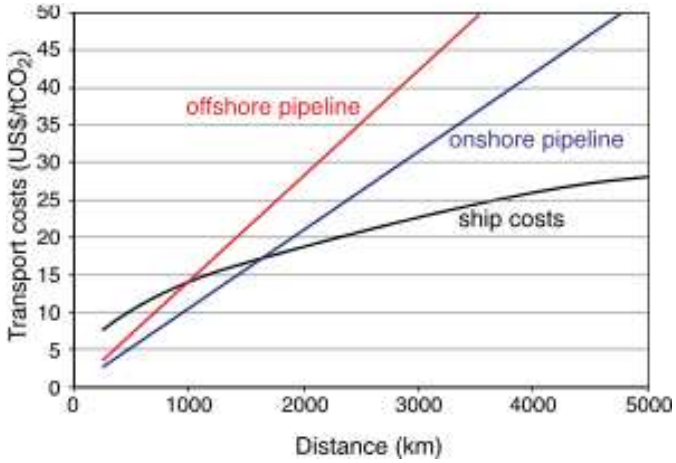


Figure 2.1: Cost analysis of CO_2 transport. Image from Coleman et al. [17].

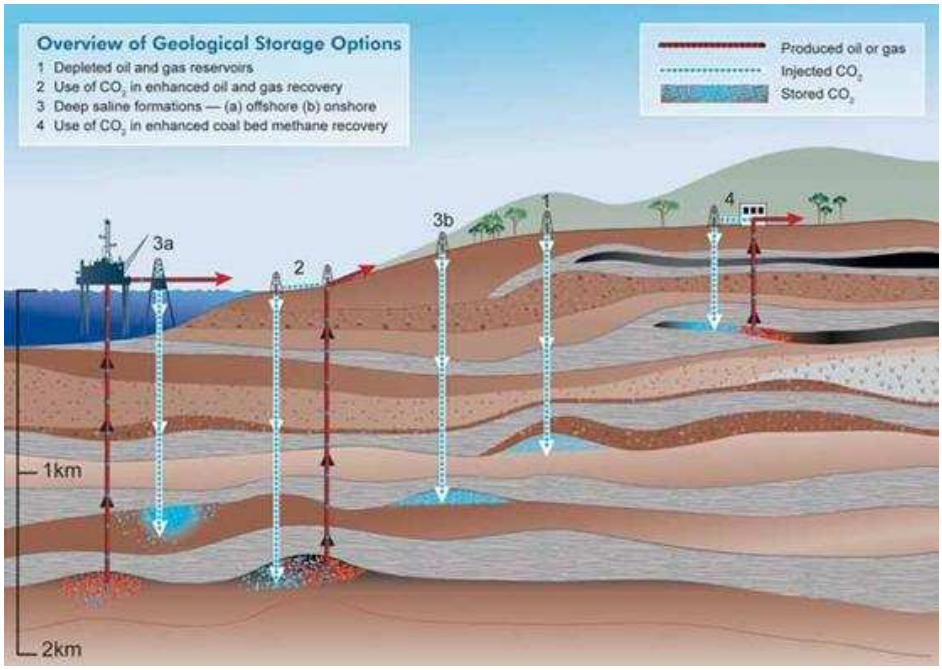


Figure 2.2: Illustration of methods for storing CO_2 in underground geological formations. Image from Anderson et al. [4].

CO_2 storage can be done in either a geological formation, like oil and gas fields, saline aquifers and coal seams, or directly in the ocean. When injecting CO_2 into the ocean, we need to inject it in depths greater than 800 meter. Over time, most of the injected CO_2 will dissolve and become part of the global carbon cycle. Both ocean storage and injection into coal seams is, to date, in a research phase and has yet to be demonstrated at a pilot scale [4, 9].

Figure 2.2 illustrates both offshore and onshore CO_2 storage. This also shows how CO_2 storage can be used in enhanced oil and gas recovery. In this thesis we will only focus on geological storage in oil and gas reservoirs below 800 meters.

2.1.1 Trapping mechanisms

Once the CO_2 is injected underground, the CO_2 must be trapped to be securely stored. The CO_2 can be subject to structural, residual, solubility as well as mineral trapping [15]. We will present all four mechanisms here, but only structural and residual trapping will be accounted for in our simulations.

Structural trapping is the most common trapping mechanisms, and means that the CO_2 percolates upward in the reservoir, since the CO_2 is more buoyant than the other liquids. Ultimately the CO_2 will reach the top of the reservoir where it meets an impermeable cap-rock, and is trapped. The CO_2 can also meet a fault, which the CO_2 can not flow through, which might trap some of the CO_2.

As the CO_2 flows through the porous rock, it displaces the fluid already present. While the CO_2 continues to flow, other fluids will replace it, but some of the CO_2 will stay behind as residual droplets in the pore space, thereby the name residual trapping. This can be compared to how water is trapped in a sponge.

When CO_2 is injected into brine, some of the CO_2 will be dissolved in the brine. This will make the dissolved CO_2 slightly heavier than the brine, and it will be trapped by sinking to the bottom. This is known as solubility trapping.

Mineral trapping is a slow process that goes on for thousands of years. When CO_2 is dissolved in water, a weak carbon acid is formed. Over time this acid can react with the minerals in the rock and form solid carbonate minerals, which effectively traps CO_2 by binding it to the rock.

2.2 Reservoir Basics

Reservoir simulation hais been performed for nearly half a century to aid the petroleum industry in deciding where one should place, and how one should operate, injection and extraction wells to optimize extraction of fossil fuels. There are several challenges in reservoir simulation, including upscaling of the rock parameters, transport models, and numerical models [1]. In this section we will focus on the basic transport and pressure model, while some upscaling using the vertical equilibrium assumption will be handled later.

2.2.1 Parameters

This part is a summarized version of an introduction given by Aarnes, Gimse, and Lie [1].

Porosity is the void volume fraction of the medium. The porosity is denoted by ϕ, and is in the range $0 \leq \phi < 1$. Since the porosity is dependent on the pressure, the rock is actually compressible. This compressibility is often neglected however, and one assumes that ϕ only depends on the spatial coordinates. For a North Sea reservoir the permeability is typically in the range $0.1 - 0.3$.

The permeability is a measure of the rock's ability to transmit a single fluid at certain conditions. The pore orientation and interconnection between pores are essential for the flow, and the permeability is strongly correlated to the porosity, but not necessarily proportional to it. The permeability is denoted $\mathbf{K}$ and is generally a tensor, however we are often able to diagonalize $\mathbf{K}$ by a change of basis. The SI-unit for permeability is m^2, but it is commonly represented in Darcy (D) or millidarcy (mD). One Darcy corresponds to about $9.869 \times 10^{-13} \mathrm{m}^2$.

The void pores in the medium are filled with different phases, like oil, water, CO_2 and brine. The volume fraction occupied by a phase is called the saturation of that phase, denoted s_i for phase i. Together the phases will occupy all the void volume, and the sum of saturations in a pore will therefore always sum to 1.

Each phase have a density (ρ) and a viscosity (μ), which are functions of the phase pressure and the composition of each phase. The density and viscosity are compressible, but the compressibility is most important for gas phases and are therefore often ignored in fluid phases. Due to interfacial tension, we also have to define a capillary pressure between phases, being the difference in density between the two phases: $p_{cij} = p_i - p_j$.

Assuming that all phases may be present at the same location, it turns out that the flow of one phase depends on the environment in that location. This means that the permeability experienced by one phase depends on the saturation of the other phases at that location, as well ass the phases' interaction with the pore walls. We therefore have to introduce a relative permeability k_i, which describes how phase i flows in the presence of the others. The effective permeability experienced by phase i in the pressence of other phases is thereby given by $k_i\mathbf{K}$.

2.2.2 Mass conservation

The incompressible transport equation describes how the saturation of a phase changes over time. One important part of this transport equation is that it is mass conservative, which means that the total volume of a phase should not change except for the volume injected or extracted of that phase [1]. One simple model for the transport of an incompressible phase i is the continuity equation:

$$\frac{\partial \rho_i \phi s_i}{\partial t} + \nabla \cdot (\rho_i \mathbf{v}_i) = q_i \tag{2.1}$$

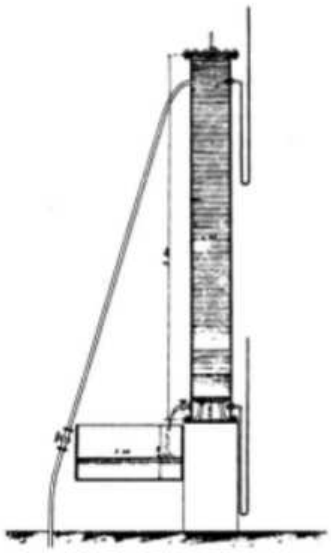


Figure 2.3: Sketch of the apparatus used by Darcy to formulate Darcy's law. Image from Darcy [19].

Here we have introduced a source term, q_i, which is the volume rate of phase i. If we define $\mathbf{v} = \sum_{i=1}^{n} \mathbf{v}_i$ to be the total flux and assume an incompressible flow, meaning the fluid density is constant, we can sum Equation (2.1) for all phases, which gives us:

$$\nabla \cdot \mathbf{v} = \sum \frac{q_i}{\rho_i} \tag{2.2}$$

The velocities will be modeled using an empirical relation called Darcy's law, after the French engineer Henry Darcy.

Darcy's law

Whereas Fourier's law describes heat conductivity and Fick's law describes diffusion, Darcy's law describes flow in a porous medium. Darcy's law was formulated by the French engineer Henry Darcy in 1856 [19]. While experimenting with water cleaning through a sand filter, Darcy found out that the filtration velocity is proportional to a combination of the gradient of the fluid pressure and pull-down effects due to the gravity, meaning the velocity is related to the pressure and gravity forces through the relation:

$$\mathbf{v}_i = -\frac{k_i}{\mu_i}\mathbf{K}(\nabla p_i - \rho_i \mathbf{g}) \tag{2.3}$$

Where p_i is the phase pressure and $\mathbf{g}$ is the gravity vector.In Figure 2.3 we see a sketch of the apparatus used by Darcy. He filled the larger tube with sand,

and injected water at the top. Then he measured the elevation of water in the two smaller tubes to the right and used this to formulate his law.

Similar to Fourier's law, which states that heat flows from warm to cold places, and Fick's law, which states that the flux goes from regions of high concentration to low concentration, Darcy's law states that the velocity goes from regions with high pressure to regions with low pressure.

Darcy's law will be used to convert (2.1) into an equation only dependent on the total velocity, as well as being used to calculate the pressure driven velocity. Assuming a two-phase, incompressible flow with equal pressure, while neglecting gravity, and defining $\lambda_i = \frac{k_i}{\mu_i}$, and $\lambda_t = \lambda_n + \lambda_w$ one can write the total velocity as:

$$\mathbf{v} = -(\lambda_n + \lambda_w)\mathbf{K}\nabla p \tag{2.4}$$

and use this to calculate the phase velocity given the total velocity by:

$$\begin{aligned} \mathbf{v}_i &= -\lambda_i \mathbf{K}\nabla p \\ &= -\frac{\lambda_i}{\lambda_t}\mathbf{K}\lambda_t \nabla p \\ &= \frac{\lambda_i}{\lambda_t}\mathbf{v} \end{aligned}$$

Putting this into (2.1) gives us the new equation:

$$\frac{\partial \rho_i \phi s_i}{\partial t} + \nabla \cdot (\rho_i \frac{\lambda_i}{\lambda_t}\mathbf{v}) = q_i \tag{2.5}$$

This will be done in more detail, without neglecting the gravity, in chapter 3.

2.3 The Vertical Equilibrium assumption

For the risk assessment needed in CO_2 storage, fast simulations are vital. The vertical equilibrium assumption, in hydrology known as the Dupuit assumption [20], is an assumption of equilibrium of flow in the vertical direction, and can be used to simplify the problem and accelerate the simulation. The assumption has long traditions in reservoir engineering, and was early on extended to handle multi-phase flow [16, 36, 37]. Equilibrium of vertical flow means that the flow upward in the reservoir is equal to the flow downward. One common misconception is that this equals to no flow in the vertical direction, but the assumption is actually equivalent to an assumption of infinite vertical flow [16].

Using this assumption, we can solve for the depth-averaged lateral flow, transforming the 3D model into a 2D model. This results in two benefits with respect to performance: Less data to compute and a looser time step restriction [34]. The decrease in data to compute on is the most obvious, as we go from 3D to

2D. The looser time step restriction is not as obvious as the decrease in data, but comes from the fact that the time step often is restricted by the coarsening of the vertical direction. The benefits of using the vertical equilibrium assumption with respect to the time step restriction has been explored by Ligaarden and Nilsen [34].

Another aspect of the risk assessment is to use reliable simulations, so we do not introduce more errors than necessary. At first glance, the removal of one dimension in the vertical equilibrium assumption may appear to introduce large errors with respect to the 3D model, but the error introduced is often less than the errors introduced by the coarse resolution needed to make the 3D model computationally tractable [34]. This means that, in many cases, a model based on the vertical equilibrium assumption is both a fast and reliable simulation tool, given the validity of the assumption.

As a typical aquifer targeted for CO_2 injection is several kilometers wide, but only 20-100 meters high, it seems natural to assume equilibrium in the vertical direction. It has been shown that the vertical equilibrium assumption is valid if the quantity $\omega = K_x/K_z(\nabla h)^2 \ll 1$ [34], which will be valid for many aquifers except for a small area around the injection well.

It has already been shown by Ligaarden and Nilsen [34], that one will gain good performance benefits, with respect to the full 3D model, by using the vertical equilibrium assumption on a model for CO_2 migration. Further in this thesis we will take advantage of the parallel architecture of the computer to accelerate the simulations further, both using the multi-core architecture of the CPU, and the extreme parallelism of the GPU.

2.4 Parallel computation

The idea of parallel computation is to split the task at hand into smaller tasks, which can be performed in parallel on separate cores, thereby taking advantage of all the cores of the machine. Highly parallel tasks, such as Monte-Carlo simulations and explicit stencil computation, can often perform many times better on highly parallel architectures.

Over the last five decades, computational performance have increased exponentially, by an almost continuous increase in clock frequency, closely following Moore's law [40]. A few years ago, this trend ceased due to physical restrictions of what silicon chips can withstand with regards to heat, and now performance increases through multiple cores at lower frequencies on the same chip. Another limitation encountered was the von Neumann bottleneck, which means that the increase in bandwidth between the CPU and the main memory have to be proportional to the performance increase, which has not happened [7]. By adding more cores on the same chip, one can gain better performance using the same amount of energy, or more performance on the same energy budget [45]. Along with more theoretical performance, you also gain fast communication between cores, as all cores reside on the same chip, and thereby share on-chip memory.

Even though standard CPUs are becoming increasingly parallel, most of the

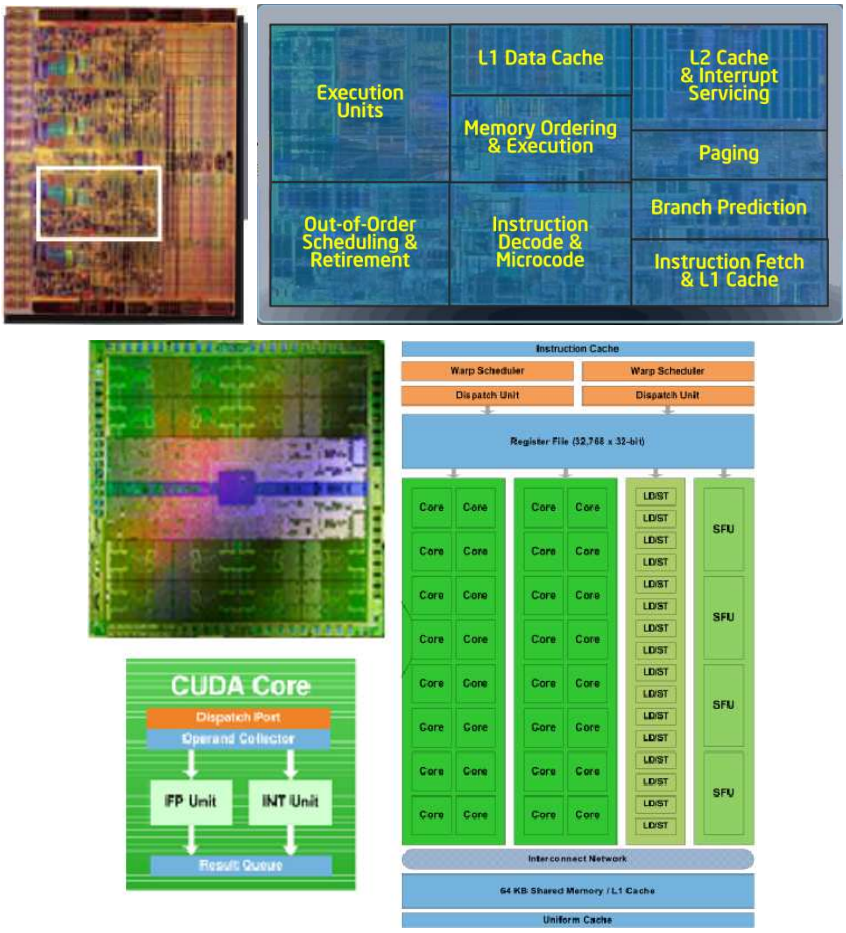


Figure 2.4: Image of the chips and cores of the newest CPU and GPU architectures. One core on the CPU is marked, and the transistors dedicated to computations are marked in the "Execution Units" section of the core. The green part of the Nvidia chip are the CUDA cores, which each have one FP Unit and one INT Unit which are dedicated for float-point and integer arithmetic respectively. Images from Intel [25] and Nvidia [44].

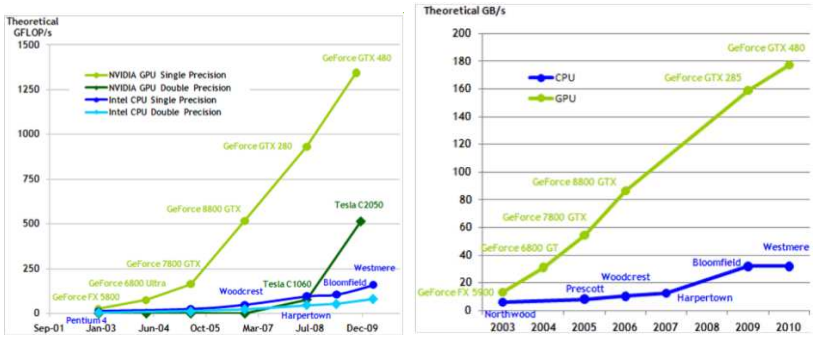


Figure 2.5: Comparison of theoretical flops and memory bandwidth between CPUs and GPUs. Image from Nvidia [44].

transistors on a CPU are still used for non-computational tasks like logic and cache. In Figure 2.4 a CPU core is compared to a GPU chip. We see that the CPU core has a lot less transistors dedicated to pure computations than the GPU. Using the multi-core processor together with one or more highly parallel accelerator units, like the GPU, gives us a heterogeneous architecture which has proven to be beneficial compared to a homogeneous architecture where a CPU did all the computational work. The accelerator units typically use many cores running at a lower frequency than the CPU, but maximise performance given a fixed power or transistor budget [11].

2.5 General-Purpose computation on Graphics Processing Units

The use of the GPU as an accelerator unit for problems not directly related to rendering, is called General-Purpose computation on Graphics Processing Units (GPGPU). These kind of problems include, but are not limited to: stencil calculations, linear algebra, audio processing, and video processing. The reason for using the GPU as an accelerator unit, is the extreme parallel architecture which is also needed for the main purpose of a GPU, 3D rendering. By transforming the algorithms used into parallel algorithms, one can take advantage of the GPU to accelerate the problem. Other hyperbolic problems, such as the shallow water equation, and single-phase flow of oil in a porous media, have previously been shown to be suitable for GPU acceleration [10, 12], so we expect that our problem will also be suitable for GPU acceleration.

The theoretical performance of GPUs and CPUs are shown in Figure 2.5. We clearly see that the GPU will outperform a CPU if we can take advantage of all the power on a GPU. One other important observation is that the GPU is not affected by the von Neumann bottleneck [7], this is because of the separate memory channel each multi-processor have to the main memory, and that the main memory is on the same chip as the processors. The GPU is also an inexpensive accelerator unit, if we compare the Nvidia Geforce GTX 480 with the fastest CPU on the Westmere architecture, Intel Core i7 Extreme 990X, the

fastest GPU and CPU in Figure 2.5, the prices are 2,595 NOK and 7,895 NOK respectively[1].

Since the GPU was originally made for 3D rendering, for which there is no need for high precision, these devices have until recently only supported single-precision floating-point numbers. This is the reason why there was no support for higher precision up until Nvidia released its GT200 architecture June 2008 [42]. The main drawback with this architecture was its double-precision performance, which was 1/4 of the single-precision performance. In march 2010, Nvidia released its GF100 architecture, best known as Fermi, which featured a double-precision performance of 1/2 of the single-precision performance, along with a fully IEEE754-2008 compliant floating-point implementation [42]. Looking at Figure 2.5, we can see this sudden increase in double-precision performance which came with the latest release.

The GPU architecture is different from vendor to vendor, and here we will only explain the architecture of Nvidia GPUs, for the newest Nvidia architecture you can use Figure 2.4 for reference. The newest Nvidia architecture consists of everything from 1 to 16 multiprocessors, which each consists of 32 cores. Previous architectures consisted of up to 30 multiprocessors, which each consists of 8 cores. These cores are the main computational units on the GPU, consisting of one floating point unit and one integer unit. The warp schedulers are the main control units of the multiprocessor, and tells each core which instructions to perform, as well as switch threads on the cores. Each warp scheduler can manage 16K registers, giving a theoretical 1024 threads for each warp, but the number of threads managed by one warp scheduler will decrease as the number of registers used increases. Theoretically, the Fermi architecture can hold up to 32,768 threads, while the GT200 architecture can hold up to 30,720 threads. The GT200 architecture also contain one double precision unit on each multiprocessor, while the Fermi architecture actually uses two cores as one double precision unit.

There are two warp schedulers for each multiprocessor on Fermi, and one on previous architectures. The newest architecture also consists of four special function units (SFU) for mathematical operations, like trigonometric functions and square roots, where the previous architectures consisted of two such units. Lastly we have load and store units, which read and writes to global memory. There are 16 such units on the Fermi architecture, while these units was placed on the core on older architectures.

Several APIs consists for taking advantage of the GPU as an accelerator unit. A few years ago, one had to use the graphics pipeline along with a 3D rendering API, like OpenGL and Direct3D, to get access to the GPU. This requires knowledge of 3D rendering, to solve problems which might have nothing to do with 3D rendering. Several attempts for making an API to the GPU without using 3D rendering APIs have been made [28, 51], but none of them reached the wanted popularity. In 2006 Nvidia released its own, platform independent, API, the Compute Unified Device Architecture (CUDA) [44], and since then the interest in GPGPU have increased both in the research and private sector. The main drawback with CUDA is the requirement of a Nvidia GPU, and in

[1] Prices from http://www.komplett.no at 2011-04-09

November 2008 the Khronos group released the specifications for OpenCL; a vendor independent alternative to CUDA [30].

In this thesis we will use CUDA as our API, and there are mainly two reasons for us not choosing OpenCL instead. One of them is that we already have a good knowledge of CUDA, and therefor do not have to learn a brand new API. Another, and more important reason, is that even though OpenCL is vendor independent, the performance is not vendor independent. OpenCL only guarantees that the program compiles and runs on different platforms, but the performance will not be the same, even though the architectures have the same theoretical performance. In OpenCL one still have to write platform specific code to gain the same performance. Because of this, we believe that using CUDA as our API will be beneficial in showing how our problem can be accelerated by using the GPU.

2.6 CUDA

In this section we will give a brief introduction to the programming model in CUDA, and how to optimize the code running on the GPU [44].

2.6.1 Programming model

Since the GPU has no operating system, CUDA uses the CPU to initialize data, copy data to the GPU, and start the GPU calculations.

Host and devices

In the CUDA programming model, the CPU controls the flow of data, while the GPU only executes small programs when told to by the CPU. This master-slave relationship is also reflected in the jargon used in CUDA, where the CPU is referred to as the host and the GPU is the device. It is also possible for the host to control several devices, but a device can only be used by one host at a time.

Kernels

Functions executed on the GPU are referred to as kernels. The kernels have access to the GPU memory areas, such as global memory, texture memory and constant memory, and data must be copied from the host memory to the device memory in order for the kernels to use the data. When a kernel finishes the results must be written back to the main memory to be available for further processing, or to be copied back to the host memory.

Copy operations

The interface provided makes it easy to move data between the GPU and the CPU, however this is an expensive operation and it is recommended to keep the number of copy operations to and from the GPU at a minimum. Alternatively one could use asynchronous data transfers, and thereby be able to continue the program flow while the data is copied in the background.

2.6.2 Thread hierarchy

When starting a kernel, the multiprocessor creates, manages, schedules and executes threads in groups of 32 threads, called a warp. These warps start the same program, but are free to branch and execute independently of each other. The warps are not part of the programing model itself, but are vital to be aware of since the blocks are made up of warps, and each block is handled by one warp scheduler, meaning the number of warps in a block is equal to the size of the block divided by 32. Each thread in a block have its own thread id, starting with 0 in the first warp. All threads in a block have access to the same shared memory space, which can be used to synchronize executions and communicate between threads. As the threads in a block have their own thread id, the blocks in a grid also have their own block id. These ids are often used to find the data a thread should work with.

The blocks form the grid, which basically only contains the dimensions in x,y and z direction of the data, as one can order the blocks and threads in both a 1D, 2D, and 3D ordering.

Along with the thread hierarchy, there is also a corresponding memory hierarchy. The memory spaces are physical spaces, and reside on either the GPU chip, the multiprocessor, or on the computational cores themselves. This means, threads on different multiprocessors, can not use the memory on the multiprocessors to communicate. There are three main memory spaces on the GPU, local memory, shared memory and global memory. The local memory reside on the computational cores, and no communication can be done in this memory space. One would actually try to not use this memory space at all, and only use the registers as, these are a lot faster than using the local memory space.

The main memory used for communication is the shared memory. This memory is on a per-block basis, so each block can use this memory space to communicate internally and synchronize the data if needed. The shared memory is cached and resides on the multiprocess, which makes it almost as fast as using the local memory space.

The last main memory space is the global memory. This is the memory space the whole grid can access, and it can be used to communicate between all threads; however, this memory space is slow and it is recommended to minimize read and write operations in the global memory space.

Figure 2.6 shows how the threads belong to a block, which again belongs to a grid. It also shows the corresponding memory hierarchy where a thread accesses the per-thread local memory, each thread in a block can access the per-block

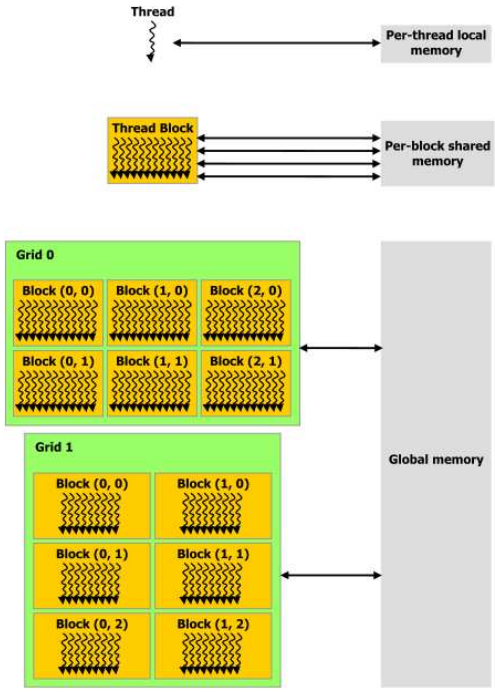


Figure 2.6: The grid and memory structure on the GPU. Image from Nvidia [44]

shared memory, and every thread can access the global memory. We also see that two different grids can access the global memory at the same time. This is something that first came with the Fermi architecture, which allowed for execution of two kernels at the same time. On GPUs released before Fermi, this was not possible, and only one grid could access the global memory at a time. A consequence of this is that one have to be careful on the newer architecture, so two grids do not work on the same set of data in global memory, as that could lead to corrupt data, as well race conditions that inhibit performance.

2.6.3 Performance optimization

Even though a straightforward GPU implementation of highly parallel problems often will give good performance gains by itself, the performance can be increased further by avoiding extensive branching, memory fetching, and memory copies [43].

Branching

Each warp started by the multiprocessor runs the same instructions simultaneously. So when one, or several, threads diverge from the rest of the threads because of some data-dependent branching, the rest of the threads in the warp

will be idle until the threads are done with the divergent branch. Idle threads will affect the throughput of data in a negative way, and one have to make sure to avoid data dependent branching as much as possible.

As the warp size is 32, which is the number of cores on a multiprocessor, it will be beneficial to keep the size of the blocks a multiple of 32. This way all the cores on a multiprocessor can work with the same block at the same time, making sure no threads are left idle.

Avoiding branches is not necessarily an easy task, as the branches depend on the data. What often is done is to calculate the outcome of all the possible branches, and then decide which should be used for this thread. This way, one will still have a branch at the end, but this is short and you will minimize the number of cycles the other threads will have to wait. On some problems one can actually remove branching all together by using max/min operations as well as bit operations instead.

Memory fetching

One of the slowest operations on the GPU itself is reading and writing to global memory, since the global memory is not cached. There are, however, cached memory spaces that can be used: Constant and texture. Both of these memory spaces are read-only, which means you will have to access the global memory space when you are storing you result, but for reading it is preferred to use either constant or texture memory.

It is also important to do coalesced memory reads and writes. This means that you align the data so the data needed for one block lays consecutive in memory. By doing this, you can read more data at the same time, up to 64 bytes at a time on architectures before Fermi and up to 128 bytes on Fermi architectures, drastically reducing the number of memory fetches and increasing the performance.

Another trick one can do to increase speed, is to write back to global memory as soon as you have calculated the data. Since the read and writes on the GPU are asynchronous, the write back to global memory will happen in the background while the threads are continuing their computations.

Memory copies

As already stated, one should minimize the number of memory copies. What this means is that data that never changes during execution, should be copied over when starting the application, and never again. This gives an overhead when starting the simulations, but minimizes the number of copy operations during the simulations. As an example, we copy the whole grid, wells, etc. to the GPU when starting the simulation and only swap the new flux and well data during the simulations.

Premature optimization

Even though CUDA is made to increase performance, the rules of code readability and performance optimization that exists for programmers, should still apply when writing GPU accelerated programs. It is, as with any other programs, highly unlikely that no one will ever touch your code again, so it is important that you write code that is possible for others to read. With regards to performance optimization it is stated: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." Knuth [32, page 268].

This does not mean optimization should not be done at all, but wait until you have a working solution before starting the optimization. Another benefit of waiting with optimizing, is that it is easy to optimize smarter. You can analyse the program by running it through a profiler, and then only optimize the bottlenecks of your code, instead of randomly optimizing the parts you believe are the slowest. Further it means that you in 97% of the cases should neglect small increases in performance, which leaves 3% where you should not neglect small increases in performance. Since we wish to run the simulations several times, a speedup of 2% will give us one extra simulation every 50th simulation, which in the long run will grow and make it possible to work more efficiently.

Chapter 3

Mathematical models and numerical simulations

In this chapter we will introduce the full 3D model of our problem. We will then use the vertical equilibrium assumption and integrate the model into a corresponding 2D equation before we derive a numerical scheme of the new model, using a finite volume method. Next, we introduce the CFL condition, before discussing different algorithms for implementing the scheme. Lastly, we will discuss in more detail the choices made to increase performance of our GPU solver.

3.1 The Vertical Equilibrium model

To derive our model, we start with the basic transport equation from Equation (2.1), along with Darcy's law for one phase flow from Equation (2.3). In our problem we are only interested in two phases, CO_2 and brine. We also neglect both rock and fluid compressibility, as well as assume a sharp interface between CO_2 and the brine. The reason we can assume that CO_2 is incompressible, is that CO_2 is actually in a liquid state at the depths we are interested in.

Assuming a two-phase, incompressible flow with equal phase pressures, and defining $\lambda_i = \frac{k_i}{\mu_i}\mathbf{K}$, $\lambda_t = \lambda_{co_2} + \lambda_w$, and $f(S) = \frac{\lambda_{co_2}}{\lambda_t}$, we can write the total velocity as:

$$\mathbf{v} = -\lambda_t \nabla p + (\lambda_{co_2}\rho_{co_2} + \lambda_w \rho_w)\mathbf{g} \tag{3.1}$$

which gives us an expression for the velocity of CO_2, given the total velocity:

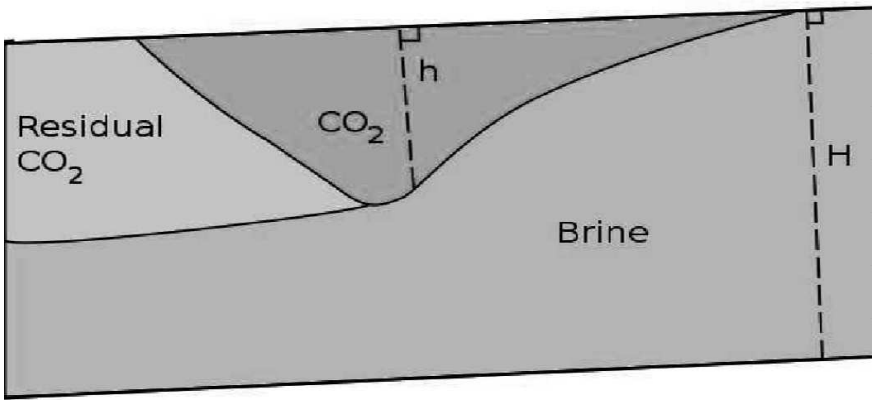


Figure 3.1: Illustration of the CO_2 plume as assumed by the VE model. Image from Ligaarden and Nilsen [34].

$$\begin{aligned}
\mathbf{v}_{co_2} &= -\lambda_{co_2}(\nabla p - \rho_{co_2}\mathbf{g}) \\
&= -f(s)(\lambda_t \nabla p - \lambda_t \rho_{co_2}\mathbf{g}) \\
&= f(s)(-\lambda_t \nabla p + \lambda_t \rho_{co_2}\mathbf{g}) \\
&= f(s)(\mathbf{v} + \lambda_w \rho_w \mathbf{g} - \lambda_w \rho_{co_2}\mathbf{g}) \\
&= f(s)(\mathbf{v} - \lambda_w \Delta\rho \mathbf{g})
\end{aligned}$$

where $\Delta\rho = \rho_{co_2} - \rho_w$. By adding and subtracting $\lambda_{co_2}\rho_w\mathbf{g}$ in Equation (3.1), and letting S be the saturation of CO_2, we can rewrite our system into:

$$\begin{aligned}
\frac{\partial \phi S}{\partial t} + \nabla \cdot f(S)(\mathbf{v} + \lambda_w(S)\Delta\rho\mathbf{g}) &= q_{co_2} \\
\mathbf{v} = -\lambda_t(\nabla p - (f(S)\rho_{co_2} + (1 - f(S))\rho_w)\mathbf{g}) & \\
\nabla \cdot \mathbf{v} &= q_{tot}
\end{aligned} \tag{3.2}$$

which is the problem we are interested in. Our focus will be on implementing the continuity equation, while Darcy's law will be used to update the pressure.

The next step is to vertically integrate Equation (3.2). To do this, we define the averaged saturation $s = h/H$ to be the relative height of the CO_2 plume as illustrated in Figure 3.1. Thus s is a function of time t and spatial coordinates $\mathbf{x}$ of the reservoir.

The vertical equilibrium assumption can be used in several different ways; where the most straight forward is a vertical averaging of the rock parameters. As we can see from Figure 3.1, the CO_2 only takes up a small portion of the height of the aquifer. This might result in large errors where the permeability is not homogeneous in the vertical direction, which easily can be seen by using the Johansen formation, where the permeability increases towards the top of the formation. This means flow would go too slow by only averaging the permeability. By slightly modifying the assumption to a vertical equilibrium in the CO_2

filled part of the aquifer; we can instead integrate the permeability in the range 0 to h, which is the permeability actually influencing the CO_2 flow. The same argument holds for the porosity, but since the porosity is relatively homogeneous, the error will not be as large. It has been shown by Ligaarden and Nilsen [34] that integrating the permeability will give a better approximation than averaging the permeability. It will also require us to calculate the permeability for every time step, meaning we increase the computational cost some.

3.1.1 Vertical Integration

For the vertical integration, we have to define a vertically integrated porosity and integrated pseudo mobilities [34]. The integrated porosity is given by:

$$\Phi(s, \mathbf{x}) = \int_0^{sH(\mathbf{x})} \phi(z, \mathbf{x})\, dz \tag{3.3}$$

while the integrated pseudo mobilities are given by:

$$\tilde{\lambda}_{co_2}(s, \mathbf{x}) = \int_0^{sH(\mathbf{x})} \frac{k_{co_2}(1)}{\mu_{co_2}} K_{||}(z, \mathbf{x})\, dz, \quad \tilde{\lambda}_w(s, \mathbf{x}) = \int_{sH(\mathbf{x})}^{H(\mathbf{x})} \frac{k_w(1)}{\mu_w} K_{||}(z, \mathbf{x})\, dz \tag{3.4}$$

These pseudo mobilities give the corresponding fractional flow functions:

$$\tilde{f}(s, \mathbf{x}) = \frac{\tilde{\lambda}_{co_2}(s, \mathbf{x})}{\tilde{\lambda}_{co_2}(s, \mathbf{x}) + \tilde{\lambda}_w(s, \mathbf{x})}, \qquad \tilde{f}_g(s, \mathbf{x}) = \tilde{\lambda}_w(s, \mathbf{x}) \tilde{f}(s, \mathbf{x}) \tag{3.5}$$

Lastly our VE equivalent of the capillary terms reads, when disregarding capillary forces in the underlying model, $p_c(s, \mathbf{x}) = H(\mathbf{x}) g_\perp \Delta\rho s$, where $g_\perp$ is the gravity component perpendicular to the surface. This gives us the new, vertically integrated model:

$$\begin{aligned} \frac{\partial \Phi(s, \mathbf{x})}{\partial t} + \nabla_{||} \cdot \left[(\tilde{f}(s, \mathbf{x}) \mathbf{v}^{ve} + \tilde{f}_g(s, \mathbf{x}) [\mathbf{g}_{||}(\mathbf{x}) + \nabla p_c(s, \mathbf{x})] \right] &= q_{co_2}(\mathbf{x}, t) \\ \nabla_{||} \cdot \mathbf{v}^{ve} &= q_{tot}(\mathbf{x}, t) \\ \mathbf{v}^{ve} = -\tilde{\lambda}_t(s, \mathbf{x}) \left[\nabla_{||} p_t - \left(\tilde{f}(s, \mathbf{x}) \rho_{co_2} + (1 - \tilde{f}(s, \mathbf{x})) \rho_w \right) \mathbf{g}_{||}(\mathbf{x}) \right] - & \\ \tilde{\lambda}_w(s, \mathbf{x}) \nabla_{||} p_c(s, \mathbf{x}) & \end{aligned} \tag{3.6}$$

To consider residual trapping, the evaluation of the relative permeabilities involves the residual saturations S_{rw} and S_{rco_2} for brine and CO_2. Then k_{co_2} is evaluated at $1 - S_{rw}$, while k_w is evaluated at $1 - S_{rco_2}$ where the historical

maximum of the average saturation is larger than the current averaged saturation. Moreover $\Phi(s, \mathbf{x})$ is multiplied by $1 - (S_{rw} + S_{rco_2})$ where the historical maximum is larger than the current averaged saturation, and $1 - S_{rw}$ elsewhere. If the porosity is constant in the vertical direction, we can simplify $\Phi(s, \mathbf{x})$ to $\phi(\mathbf{x}) \cdot sH(\mathbf{x})$. Using this, the transport equation takes the simpler form:

$$\phi(\mathbf{x})\frac{\partial H(\mathbf{x})s}{\partial t} + \nabla \cdot \left[\tilde{f}(s, \mathbf{x})\mathbf{v}^{ve} + \tilde{f}_g(s, \mathbf{x})[\mathbf{g}_{||}(\mathbf{x}) + \nabla p_c(s, \mathbf{x})]\right] = q_{co_2}(\mathbf{x}, t) \quad (3.7)$$

To simplify notations in the next section, we define the total flux as:

$$\mathbf{f}(s, \mathbf{x}) := \tilde{f}(s, \mathbf{x})\mathbf{v}^{ve} + \tilde{f}_g(s, \mathbf{x})[\mathbf{g}_{||}(\mathbf{x}) + \nabla p_c(s, \mathbf{x})]$$

3.2 The Finite Volume Method

In this section we will use the finite volume method to derive a scheme which can be used to solve our model problem in Equation (3.7). This means partitioning the model into cells, and integrating over these cells. Looking at cell $C_{i,j}$ we get:

$$\int_{C_{i,j}} \phi(\mathbf{x})\frac{\partial H(\mathbf{x})s(\mathbf{x}, t)}{\partial t}\, d\mathbf{x} + \int_{C_{i,j}} \nabla \cdot \mathbf{f}(s, \mathbf{x})\, d\mathbf{x} = \int_{C_{i,j}} q_{co_2}(\mathbf{x}, t)\, d\mathbf{x} \quad (3.8)$$

If we assume the size of the cells tends towards zero, we can also assume that the porosity is constant inside the cell.

$$\phi_{i,j} \int_{C_{i,j}} \frac{\partial H(\mathbf{x})s(\mathbf{x}, t)}{\partial t}\, d\mathbf{x} + \int_{C_{i,j}} \nabla \cdot \mathbf{f}(s, \mathbf{x})\, d\mathbf{x} = \int_{C_{i,j}} q_{co_2}(\mathbf{x}, t)\, d\mathbf{x} \quad (3.9)$$

Integrating Equation (3.9) in time, we get an expression for the next time step:

$$\begin{aligned}\phi_{i,j} \int_{C_{i,j}} H(\mathbf{x})(s(\mathbf{x}, t_{n+1})) - s(\mathbf{x}, t_n))\, d\mathbf{x} + \int_{t_n}^{t_{n+1}} \int_{C_{i,j}} \nabla \cdot \mathbf{f}(s, \mathbf{x})\, d\mathbf{x}\, dt \\ = \int_{t_n}^{t_{n+1}} \int_{C_{i,j}} q_{co_2}(\mathbf{x}, t)\, d\mathbf{x}\, dt\end{aligned} \quad (3.10)$$

The integration of the flux in space, can be written as:

$$\begin{aligned}\int_{C_{i,j}} \nabla \cdot \mathbf{f}(s,\mathbf{x})\, d\mathbf{x} = &\int_{x_{i-1/2}}^{x_{i+1/2}} \mathbf{f}(s,x,y_{j+1/2}) - \mathbf{f}(s,x,y_{j-1/2})\, dx \\ &+ \int_{y_{j-1/2}}^{y_{j+1/2}} \mathbf{f}(s,x_{i+1/2},y) - \mathbf{f}(s,x_{i-1/2},y)\, dy\end{aligned} \tag{3.11}$$

Now we simplify the equations further, by defining the averaged saturation in a cell $S_{i,j}$, the face fluxes $\mathcal{F}$ and $\mathcal{G}$, as well as the total volume rate of source in a cell $\mathcal{Q}$ by:

$$\begin{aligned}S^n_{i,j} &= \frac{1}{|C_{i,j}|}\int_{C_{i,j}} H(\mathbf{x})s(\mathbf{x},t_n)\, d\mathbf{x} \\ \mathcal{F}^n_{i,j} &= \frac{1}{|C_{i,j}|}\int_{t_n}^{t_{n+1}}\int_{x_{i-1/2}}^{x_{i+1/2}} \mathbf{f}(s,x,y_j)\, dx \\ \mathcal{G}^n_{i,j} &= \frac{1}{|C_{i,j}|}\int_{t_n}^{t_{n+1}}\int_{y_{j-1/2}}^{y_{j+1/2}} \mathbf{f}(s,x_i,y)\, dy \\ \mathcal{Q}^n_{i,j} &= \frac{1}{|C_{i,j}|}\int_{t_n}^{t_{n+1}}\int_{C_{i,j}} q_{co_2}(\mathbf{x},t)\, d\mathbf{x}\end{aligned}$$

These definitions simplify our numerical scheme into:

$$S^{n+1}_{i,j} = S^n_{i,j} - \frac{1}{\phi_{i,j}}\left[\mathcal{F}^n_{i,j+1/2} - \mathcal{F}^n_{i,j-1/2} + \mathcal{G}^n_{i+1/2,j} - \mathcal{G}^n_{i-1/2,j} - \mathcal{Q}^n_{i,j}\right] \tag{3.12}$$

As we can see from the equation, the change in cell saturation only depends on the source inside the cell itself and the fluxes over the cell faces. This is, as it should be, a discrete equivalent to Equation (3.7), which states that the change at a point in the reservoir only depends on the sources at that point as well as on the fluxes into and out of that point.

Up to this point, no approximation except for discretization of the domain has been done, which means that this equation is exact as the size of the cells tends towards zero. From here on out, however, we will have to approximate the face fluxes as well as the time integrals to get a fully explicit scheme.

The first step in making an explicit scheme is to approximate the time integrals by the value at the previous time step. This means using:

$$\int_{t_n}^{t_{n+1}} f(t)\, dt \approx \Delta t f(t_n)$$

Alternatively we could have used the same time step as we are solving for, which would have made an implicit scheme, or a Gaussian quadrature rule to approximate the integral, which would make a Runge-Kutta time-stepping

scheme. The implicit scheme would need an iterative method like the Newton-Raphson method to solve the corresponding discrete nonlinear system, which is not that easy to transform into a parallel algorithm, this is the reason why we stick to an explicit scheme. In recent publications, however, the implicit scheme has been used with success where the GPU was used both to calculate the preconditioner and solve the linear system [5, 31].

Our change over a face is given by:

$$\mathcal{F}^n_{i,j-1/2} = \int_{x_{i-1/2}}^{x_{i+1/2}} \tilde{f}_{i,j-1/2} \left[\mathbf{v}^{ve}_{i,j-1/2} + \tilde{\lambda}^w_{i,j-1/2} (\mathbf{g}_{||_{i,j-1/2}} + \nabla p_{c_{i,j-1/2}}) \right] \quad (3.13)$$

Since the velocity and gravity are calculated on a per face basis, the problem lay in finding an approximation to the fractional flow rate and the pseudo mobilities. Since the fractional flow rate is expressed from the pseudo mobilities, we only have to find an expression for the pseudo mobilities over a face.

3.2.1 The Upstream Mobility Weighting Scheme

The upstream mobility weighting scheme assumes that the flow over a cell only travels in one direction, and the mobility over a face is only dependent on the mobilities over the two neighboring cells. We also assume a piecewise constant saturation over each cell. This gives the face pseudo mobilities:

$$\tilde{\lambda}^{co_2}_{i,j-1/2} = \begin{cases} \tilde{\lambda}^{co_2}_{i,j} & \text{if } (\mathbf{v}^{ve} > 0 \text{ and } (\mathbf{g}_{||} + \nabla p_c) > 0) \text{ or} \\ & \quad (\mathbf{v}^{ve} - \tilde{\lambda}^w_{i,j-1/2}(\mathbf{g}_{||} + \nabla p_c) > 0) \\ \tilde{\lambda}^{co_2}_{i,j-1} & \text{else} \end{cases} \quad (3.14)$$

$$\tilde{\lambda}^{w}_{i,j-1/2} = \begin{cases} \tilde{\lambda}^{w}_{i,j} & \text{if } (\mathbf{v}^{ve} > 0 \text{ and } (\mathbf{g}_{||} + \nabla p_c) > 0) \text{ or} \\ & \quad (\mathbf{v}^{ve} - \tilde{\lambda}^{co_2}_{i,j-1/2}(\mathbf{g}_{||} + \nabla p_c) > 0) \\ \tilde{\lambda}^{w}_{i,j-1} & \text{else} \end{cases} \quad (3.15)$$

This also holds for $\tilde{\lambda}_{i,j+1/2}$, $\tilde{\lambda}_{i-1/2,j}$ and $\tilde{\lambda}_{i+1/2,j}$ and will be used to calculate the fractional flow across the face.

3.3 The CFL condition

The CFL condition is a necessary condition which must be satisfied for the numerical method to be stable, and thereby converge to the solution as the grid is refined.

CFL Condition [33]: *A numerical method can be convergent only if its numerical domain of dependence contains the true domain of dependence of the PDE, at least in the limit as Δt and Δx go to zero.*

Our domain of dependence, using the upstream mobility weighting scheme, is just the neighboring cells. So we have to make sure that our flux is not transported further than the cell itself, meaning we have the CFL condition:

$$\frac{\bar{u}\Delta t}{\Delta x} \leq \phi \tag{3.16}$$

where $\bar{u}$ is the distance traveled by the CO_2. The distance traveled by the CO_2 is determined by the gravity-driven velocity given by $[\mathbf{g}_{||}(\mathbf{x}) + \nabla p_c(s, \mathbf{x})]_x \frac{\partial}{\partial S}\tilde{f}_g$ and the pressure driven velocity given by $\mathbf{v}_x^{ve}\frac{\partial}{\partial S}\tilde{f} + q$. This gives the CFL condition:

$$\Delta t \leq \frac{\phi \Delta x}{2\max\left([\mathbf{g}_{||}(\mathbf{x}) + \nabla p_c(s, \mathbf{x})]_x \frac{\partial}{\partial S}\left[\tilde{f}_g\right], \mathbf{v}_x^{ve}\frac{\partial}{\partial S}\left[\tilde{f}\right] + q\right)} \tag{3.17}$$

which also holds for the y-direction. It is important to note that the CFL condition is only a necessary condition for stability, and it does not guaranty it. The condition given in Equation (3.17) will not be sufficient if the parabolic part of the equation is the dominating part for migration, which means special care have to be taken if one sees that the time step gives rise to instabilities. It has been shown by Ligaarden and Nilsen [34] that, at least for the Johansen formation, the segregation will give the dominating time step restriction, which is the part we account for in this thesis.

3.4 Implementation

In this section we will describe an algorithm for solving (3.12) using the upstream mobility weighting scheme from (3.14) and (3.15). We will start by describing a standard algorithm for solving this equation in a serial program, and then do a few changes to get a more suitable algorithm for a parallel solver. Lastly, we will take a closer look at the GPU implementation and the different choices that we made to optimize the GPU solver.

3.4.1 Algorithm

Our finite volume scheme in Equation (3.12) requires us to loop over each face to calculate the change this face will contribute to for the two neighboring nodes. We also have to account for the source term, which means we have to loop over each cell to calculate the change made from the source term. An algorithm for this can be seen in Algorithm 1. This algorithm accesses each face only once, while each cell is accessed a maximum of five times, once to calculate the change with respect to the source term, and one time for each of the four faces of a cell.

This algorithm is not really suitable for parallel execution. The reason for this is that we change two cells for each face, and each cell has four faces. This means that we would most likely access the same cell at the same time in different threads, giving writes to the same memory block and possible race conditions.

Algorithm 1 The standard serial algorithm for solving one time step of our problem.

for all c in cells **do**
  if $c.h_{max} > c.h$ **then**
    $c.\Phi = c.\Phi(1 - (S_{rw} + S_{rco_2}))$
  else
    $c.\Phi = c.\Phi(1 - S_{rw})$
  end if
  $c.h = c.h - \Delta t(max(c.q, 0) - min(c.q, 0)c.\tilde{f})/c.\Phi$
end for
for all f in faces **do**
  if f **not** boundary face **then**
    Cell $c1$ = f.neighbor[0];
    Cell $c2$ = f.neighbor[1];
    f.mob = faceMobility($c1$,$c2$)
    $\tilde{f} = f.\text{mob}_{co_2}/(f.\text{mob}_{co_2} + f.\text{mob}_w)$
    $c1.h = c1.h + \Delta t\tilde{f}(\mathbf{v}^{ve} + f.\text{mob}_w(g_{||} + \nabla p_c))/c1.\Phi$
    $c2.h = c2.h - \Delta t\tilde{f}(\mathbf{v}^{ve} + f.\text{mob}_w(g_{||} + \nabla p_c))/c2.\Phi$
    $c1.h_{max} = max(c1.h, c1.h_{max})$
    $c2.h_{max} = max(c2.h, c2.h_{max})$
  end if
end for

Algorithm 2 Function for computing the face mobility.

Function faceMobility(Cell $c1$, Cell $c2$)
  if sign($\mathbf{v}^{ve}$) == sign($g_{||} + \nabla p_c$) **then**
    if $\mathbf{v}^{ve} > 0$ **then**
      $\text{faceMob}_{co_2} = c1.\text{mob}_{co_2}$
    else
      $\text{faceMob}_{co_2} = c2.\text{mob}_{co_2}$
    end if
    if $\mathbf{v}^{ve}$−$\text{faceMob}_{co_2}(g_{||} + \nabla p_c) > 0$ **then**
      $\text{faceMob}_w = c1.\text{mob}_w$
    else
      $\text{faceMob}_w = c2.\text{mob}_w$
    end if
  else
    if $\mathbf{v}^{ve} > 0$ **then**
      $\text{faceMob}_w = c1.\text{mob}_w$
    else
      $\text{faceMob}_w = c2.\text{mob}_w$
    end if
    if $\mathbf{v}^{ve}$+$\text{faceMob}_w(g_{||} + \nabla p_c) > 0$ **then**
      $\text{faceMob}_{co_2} = c1.\text{mob}_{co_2}$
    else
      $\text{faceMob}_{co_2} = c2.\text{mob}_{co_2}$
    end if
  end if
  return faceMob

This can be fixed by using locks on the cells, so no thread can write to a cell while it is accessed by another thread. This would fix the race condition problems, but would force some threads to wait for other threads before they could continue, thereby giving an overhead to the execution.

An atomic function performs a read-modify-write operation in global memory on the GPU, and guarantees that no other threads can access this memory address until the operation is complete. We could have taken advantage of atomic functions to implement Algorithm 1 in a thread safe manner. The downside of this is that we would have more random accesses to the global memory, which implies we would not have coalesced memory reads and writes. As non-coalesced memory writes are a lot slower than coalesced, we would like to avoid the use of atomic functions.

Another drawback is that using atomic add functions for floating point data is not supported for architectures of compute capability less than 2.0, which means you need a Fermi card for these functions. On older GPUs, you can make your own atomic add for floating point numbers, using the other atomic functions. This requires even more random access to global memory and will cause your GPU kernel to be even slower. It is clear that Algorithm 1 is not a good choice of algorithm, and we will propose an alternative.

Algorithm 3 A thread safe algorithm for solving one time step of our problem.

for all c in cells **do**
  if $c.h_{max} > c.h$ **then**
    $c.\Phi = c.\Phi(1 - (S_{rw} + S_{rco_2}))$
  else
    $c.\Phi = c.\Phi(1 - S_{rw})$
  end if
  $dz = -(max(c.q, 0) + min(c.q, 0)c.\tilde{f}$
end for
for all f in faces **do**
  if f **not** boundary face **then**
    Cell $c1$ = f.neighbor[0];
    Cell $c2$ = f.neighbor[1];
    f.mob = faceMobility($c1$,$c2$)
  end if
end for
for all c in cells **do**
  for all f in $c.faces$ **do**
    $\tilde{f} = f.\text{mob}_{co_2}/(f.\text{mob}_{co_2} + f.\text{mob}_w)$
    if $c1 == c$ **then**
      $dz = dz + \tilde{f}(\mathbf{v}^{ve} + f.\text{mob}_w(g_{||} + \nabla p_c))$
    else
      $dz = dz - \tilde{f}(\mathbf{v}^{ve} + f.\text{mob}_w(g_{||} + \nabla p_c))$
    end if
  end for
  $c.h = c.h - \Delta t \cdot dz/c.\Phi$
end for

To solve our race condition issues without sacrificing performance, a rewrite of the algorithm is needed. Instead of looping over each face and calculate the change in the two neighboring cells, we could loop over each cell and calculate the contributions to one cell from each of the four faces. This approach can be seen in Algorithm 3 which we can see will do exactly the same as Algorithm 1, but without the race conditions. Further we see that this will only access each cell once, but access each face twice as one face is shared by two cells.

Assuming that n is the number of cells in our grid, and assume $n \to \infty$, we can assume that there are few boundary cells and faces. Since each cell has four faces and each face has two cells, we can assume that the number of faces will go towards $2n$ as the number of cells increases. Using this, it is easy to calculate the number of flops needed in a worst case scenario. Algorithm 1 will need a total of $40n$ flops, while Algorithm 3 will need a total of $36n$ flops, both while neglecting the flops for computing the mobilities for each face. This means that Algorithm 3 should be faster both when being executed on one core and on multiple cores. In addition 3 is thread safe without sacrificing performance, which is a huge benefit with regards to a parallel algorithm.

3.4.2 GPU optimizations

One of the biggest problems encountered when implementing on the GPU was the use of shared memory. As we see from Algorithm 3 we will calculate the change over a face twice, but using the shared memory we can get away by computing it once, store it in shared memory and fetch it from shared memory next time it is to be computed. In our implementation this was not that straightforward, since the implementation is used by MRST (MATLAB Reservoir Simulation Toolbox) [38] where everything is stored in 1D arrays. The structure used makes it easy to write solvers and find connections, but requires several memory fetches to find out where the neighboring cells and faces resides in memory.

To take advantage of the shared memory space, one could use two lookup tables. One table store the face index, while the other table store the change in CO_2 height over that face. The index we store the data in is ($faceidx$ mod $tablesize$), meaning that if we have a table size of 101, and should store the change over face number 1032, the index in the table is $(1032 \mod 101) = 22$. We then save the number 1032 in the table that stores the face index, and the change in the other table. The next time we should find the change over face 1032 we only have to check the tables, instead of actually calculate the change again. Since also 1133 will be saved in index 22, we need the first table so we do not use the change over face 1032 as the change over face 1133. To try avoid these kind of collisions, the size of the tables are a lot bigger than the size of the block, as well as being a prime number. Another drawback is that the two cells which share a face is not guaranteed to be calculated in the same block, resulting in lookup misses. The only way to avoid this is to restructure the whole grid into a 2D grid with direct access to neighbors, where the restructuring itself would give a high overhead in time consumption as well as increased memory requirements. Another problems is that this will not work on general grids, meaning we would drastically reduce the number of grids we could use.

Listing 3.1: Routine for computing face mobilities without branching

```
// Store the cell mobility in a four index array
float faceMob[2];
float cmob[4];

float2 tmpmob = mob[neighbor0];
cmob[0] = tmpmob.x;
cmob[1] = tmpmob.y;
tmpmob = mob[neighbor1];
cmob[2] = tmpmob.x;
cmob[3] = tmpmob.y;

// Get which phase we are computing.
int phase = signbit(g_flux*d_flux);
// Get which cell to take the value from.
int cell = signbit(d_flux);
faceMob[phase] = cmob[2*cell + phase];

// Check which phase we have computed.
int sign = 2*signbit(g_flux*d_flux) - 1;
// Get the cell to compute for next.
cell = signbit(d_flux + sign*faceMob[phase]*g_flux);
// Change phase.
phase = 1 - phase;
faceMob[phase] = cmob[2*cell + phase];
```

Another problem encountered was the extensive branching needed to calculate the face mobility, as well as the branching needed to calculate the pore volume. Use of bit and max/min operations on these parts of the algorithm made sure we could write this code without any other branches than the loops needed as well as the branch made when introducing the shared lookup tables. The lookup table branch will give some overhead, but the use of shared memory instead of global memory proved to be more beneficial than avoiding that branch. The loops could be manually unrolled to avoid those branch, but this is not necessary as the CUDA compiler will unroll them for us and make sure our source code still is readable. The routine for calculating the face mobilities are given in Listing 3.1, where we can see that no branches are present in the code.

By profiling the simulator we saw that a big part of the GPU time was spent fetching data from memory. To reduce the time spent fetching data one could move the static data to the texture memory space, which is cached and therefore a lot faster than the global memory. This does not reduce the number of memory fetches, but reduces the time spent reading memory drastically.

Chapter 4

Results and Discussion

In this chapter we will give an overview of the testing rigs used, and discuss the choice of using two different rigs. We will then show the test results obtained using different grids, and discuss the results obtained.

4.1 Hardware and testing

In this thesis we have decided to use two different testing rigs. We have used one rig to test the performance of the GPU solver, and a different rig for testing the CPU solver. The reason for using two rigs is to gain access to 16 CPU cores, and thereby being able to test the multi-core solver using more cores. The rig used for testing the single and multi-core CPU solver is given in Table 4.1, while the rig for testing the GPU implementation is given in Table 4.2 with the GPU-specific info in Table 4.3. Clearly the theoretical performance on the CPU testing rig is a lot higher than the theoretical performance of the CPU on the GPU testing rig, while the GPU itself have approximately 6.6 times the theoretical performance of the CPUs on the CPU testing rig.

4.1.1 Test setup

Unless stated otherwise, each test represents 400 years of simulation with 150 years of injection, where each simulation has been repeated several times. The tests are performed by updating the pressure for every year of simulation, while the time step for the transport satisfies the CFL condition (3.17). All simula-

Table 4.1: Hardware specifications for the CPU testing

CPU	Intel Xeon E7440 @ 2.40GHz 4 processors with 4 cores each
Ram	16 Gb DDR2 RAM
Theoretical flops	153.6 GFlops

Table 4.2: Hardware specifications for the GPU testing

CPU	Intel i7 920 @ 2.67GHz 4 cores
Theoretical flops	42.72 GFlops
GPU	GeForce GTX 275
Ram	4Gb DDR3 RAM

Table 4.3: GPU specifications

Core clock	1.40GHz
Number of multiprocessors	30
Numer of cores	240
Theoretical flops	1010.88 GFlops
Bandwidth	127 GiB/s
Global memory	896 MB
Shared memory	16 KB pr. block
Maximum block size	$512 \times 512 \times 64$
Maximum grid size	$65535 \times 65535 \times 1$
Max threads pr. block	512
Registers pr. block	16384

tions will be run using no-flow boundary conditions, which is currently the only boundary condition implemented.

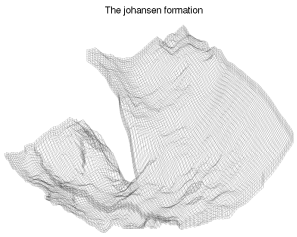
The fluid model used are similar to what is used in several benchmark studies of CO_2 injection [14, 18, 21, 34]. The fluid properties are reference values for CO_2 and brine taken at 300 bar. At this pressure the approximate viscosity and density of supercritical CO_2 is 0.057 cP and 686.54 kg/m^3 respectively, and 0.3086 cP and 975.86 kg/m^3 for the brine.

The numbers are taken from the fastest simulation, and is the mean value of simulating one year. Mean values for the whole simulation, the injection phase and the post injection phase are given to show how our simulators scale with the inclusion of source terms, which will change the time step.

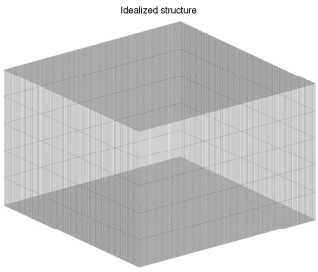
The grids used in the tests can be seen in Figure 4.1.

4.1.2 Note

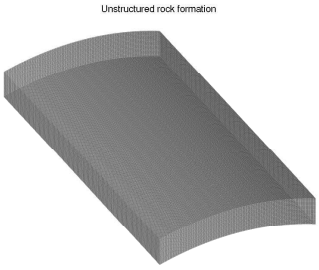
An error in the code caused many of the first GPU tests to actually run on the CPU. This was discovered a few days before the deadline of the thesis, and hence we had to revert to an older version, which did not include all optimizations that were implemented in the erroneous code. The main drawbacks of the GPU implementation used in the following tests, is that we do not use shared or texture memory, and that the calculation of the time step restriction is done on the CPU. We have, however, minimized the number of copies to and from the GPU and removed branching inside the CUDA kernels where possible. The results will therefore give a good indication as to how suitable this problem is for GPU acceleration, and for a more optimized code we expect even better



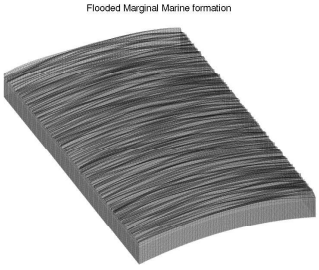
(a) The johansen formation



(b) An idealized grid (Scales not representative)



(c) Synthetic rock structure with a unstructured top surface



(d) Synthetic rock structure with a structured top surface

Figure 4.1: Plot of the grids used in the tests.

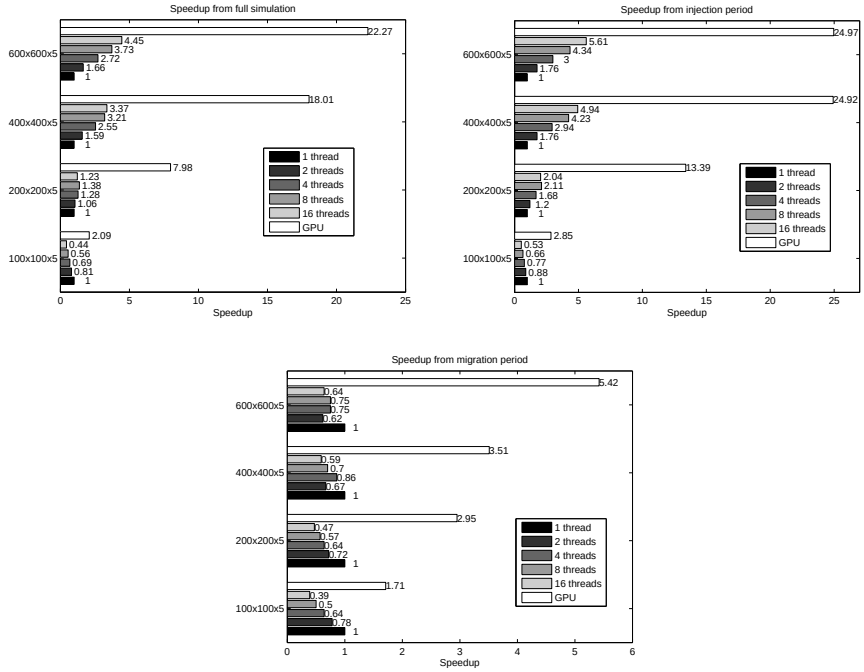


Figure 4.2: Speedup obtained by increasing the horizontal resolution.

performance than what is reported in the following.

4.2 Idealized grids

The grids used in these tests are rectangular grids which is 40x30 kilometers in the horizontal directions and 100 meters high. CO_2 is injected at the point $[15, 15]$ km, at a rate of $1,400$ m^3/day. The porosity of the rock is set to a constant at 0.3, while we have a homogeneous permeability between 10 and 1000 mD.

4.2.1 Horizontal resolution

For the test of horizontal resolution we set the vertical resolution constant, while the horizontal resolution was refined.

The results are shown in Figure 4.2. We see that the full simulation and the injection period behaves almost the same, while the migration period has a negative speedup while we increase the number of CPUs, and a far worse speedup than the injection phase for the GPU. The reason for this behavior in the migration phase, is that the time step restriction for the transport is a lot looser in the migration phase. On the 200x200x5 grid, we do 28 transport steps for every year in the injection phase, and only four transport steps for every year in the

migration phase. This means we have less transportation steps for each pressure update in the migration phase, than in the injection phase, and the creation of threads, as well as the data copy to and from the GPU, will therefor give a noticeable overhead to the simulation. One solution to this problem might be to not update the pressure as often in the migration phase of the simulation, which means we do not have to make threads and copy data to the GPU as often. This possibility will be explored later on.

For the injection time, we see that we get a negative speedup for the grid with the smallest resolution on the CPU, this is caused by the small amount of data to compute on, and so the overhead of creating threads will again be noticed. We also see a quite small speedup for the GPU on the same resolution, which is due to the small data set and the overhead of moving data. For the two grids with highest resolution, we can see a logarithmic speedup with regards to the number of cores used. This means that we, at some point, will no longer see a speedup when introducing more cores. We further see that the maximum speedup for this idealized grid, seems to converge to something around 25 for the GPU solution, and that the speedup increases a lot while we increase the resolution, up until the two grids with highest resolution. This might indicate that we have reached the peak performance for our implementation, at least when updating the pressure every year.

4.3 Vertical resolution

The setup for this test is exactly the same as for the previous test, but now we keep the horizontal resolution constant at 200x200, and refine the vertical resolution.

From Figure 4.3 we see that the simulator does not scale as well for the vertical resolution as it does for the horizontal resolution. The reason for this is that most of the work is only affected by the horizontal resolution, while only the vertical integration is affected by the vertical resolution. The vertical integration is not as expensive to do as the rest of the simulations on the CPU, and only takes up $8 - 20\%$ of the main loop, which is probably the reason why we do not gain the same speedup. For the GPU solver, however, it might involve divergent branches and uncoalesced memory reads. Because of this we have actually witnessed that the mobility calculation can take up to 88% of the GPU time, on the synthetic grid formation with a structured top surface used later.

The multi-threaded solver seems to behave better than the GPU solver, as it is not affected by uncoalesced memory reads. We do, however, see the same trend for this multi-core solver, that the speedup by increasing the vertical resolution is not as good as when we increase the horizontal resolution.

4.4 Synthetic test suite.

These tests will show how the solvers scale with respect to the addition of changes in the top surface of the rock, as well as both uniform and non-uniform

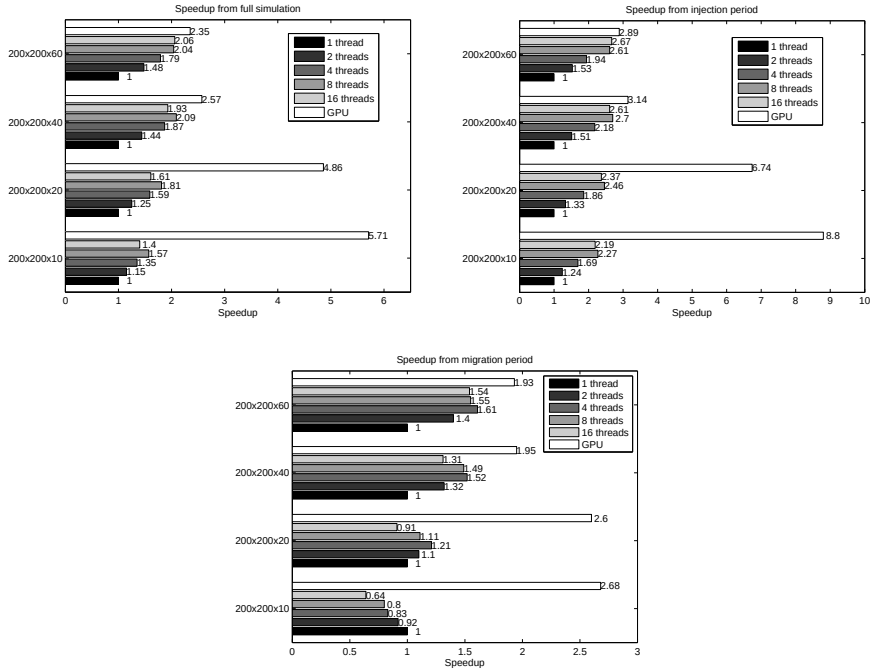


Figure 4.3: Speedup obtained by increasing the vertical resolution.

faults in the formation. There are four different fault structures introduced:

- UP1 - Uniform parallel faults.
- UP2 - Uniform parallel faults, two intersecting sets.
- NP1 - Non uniform parallel faults.
- NP2 - Non uniform parallel faults, two intersecting sets.

The formation is 30x60km wide, and 100m thick, and has an angle between top and bottom of approximately 0.5 degrees. We have used both a formation with no structure, called flat, and a formation with a structured top surface, called FMM, which both have a resolution of 300x600x20 grid cells. We inject CO_2 at the point $[15, 15]$ km, at a rate of $1,400$ m^3/day. The porosity in the formation is in the range $[0.2, 0.25]$, while the permeability is between 0.2 and 1 Darcy.

Figure 4.4 shows the change in speedup by introducing faults in a flat reservoir. We see that there is not much changes in speedup by introducing faults, neither uniform nor non-uniform faults in the injection period, this holds for both the multi-core solver, and the GPU solver. Looking at the actual simulation times, there is no noticeable difference in total time spent when introducing faults in the unstructured reservoir. As for the idealized grid, we see that the speedup is a lot worse for the migration phase than for the injection phase, and we see a maximum speedup on the GPU of 3.79, and only 1.53 for the multi-core solution. This is again caused by the time step restrictions in the migration phase.

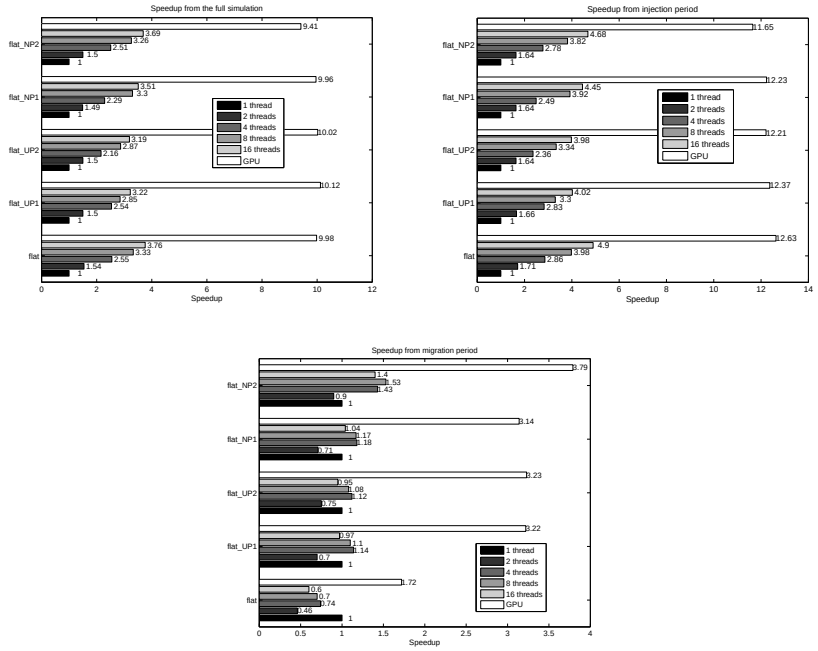


Figure 4.4: Speedup obtained by introducing faults on an unstructured rock formation.

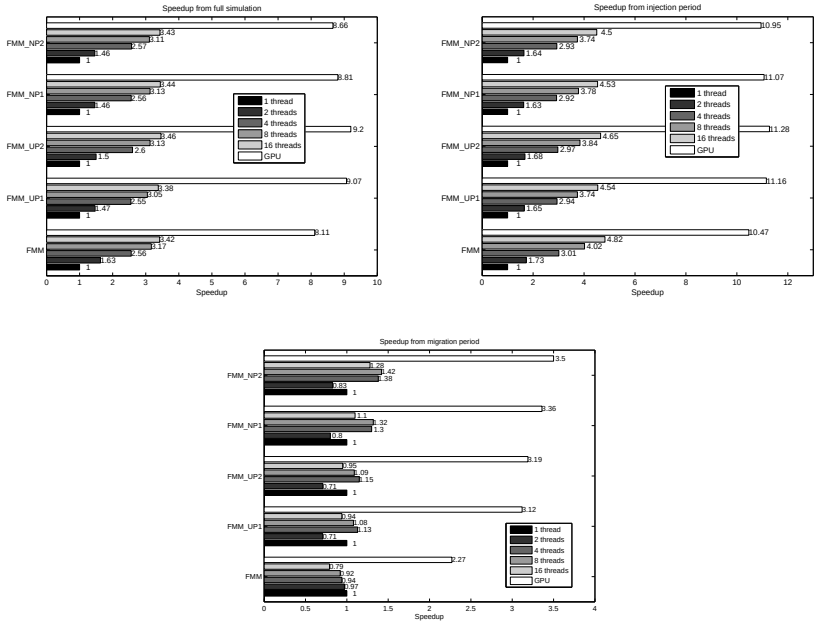


Figure 4.5: Speedup obtained by introducing faults on a structured rock formation.

In Figure 4.5, which is the speedup for our flooded marginal marine formation, we see that the same holds for this formation. We do also see that the speedups obtained, are not as good as for the unstructured formation. This especially holds for the GPU solution, and we can say that the inclusion of structures in the top surface will affect the multi-core and GPU solution in a negative way.

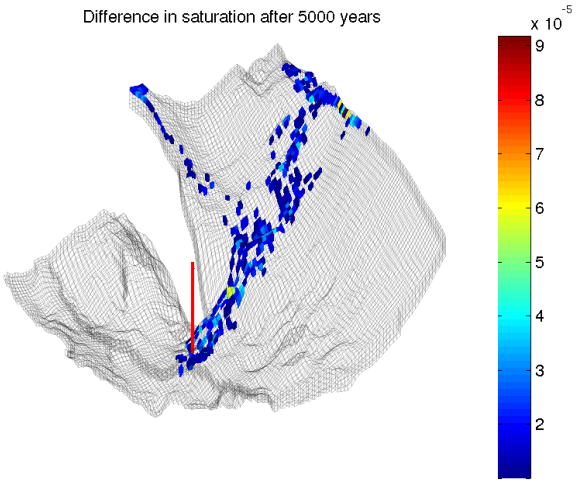
4.5 Accuracy

To test the accuracy of using single precision floating-point numbers instead of double precision floating-point numbers, we have used a model of the Johansen formation as given by the MatMoRA project [39]. The simulation represents 50 years of injection, and 4950 years of migration, giving a total of 5000 years of simulation. The error given is the relative error of the height, meaning we have scaled the difference by the largest possible error given the double precision height: $\max(H-h, h)$, where H is the height of the reservoir and h is the height of CO_2 for the double-precision solution.

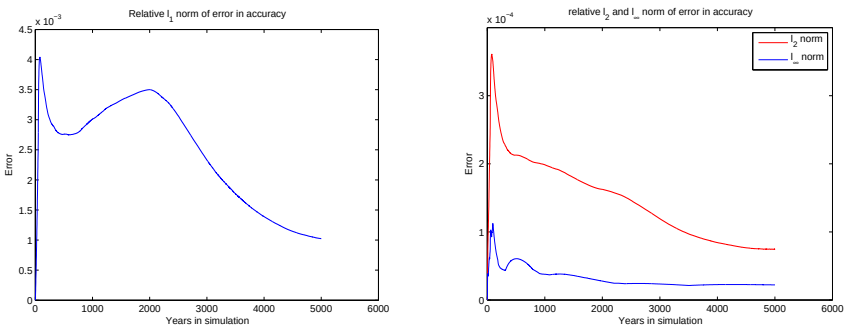
As we can see from Figure 4.6(a) there will be no visual difference in the results obtained using single precision compared to the results obtained using double precision, even after running 5000 years of simulation. The relative norm of the difference from Figure 4.6(b) is taken from the difference in height, not of the saturation. We see that after around ~ 50 years of simulation the derivative of the error decreases. Running more simulations shows that this decrease in derivative actually happens when we stop injecting CO_2. This is expected as the inclusion of a source term decreases the time step, which forces us to do more calculations for each year of simulation and thereby decreasing the accuracy.

Another interesting observation is that the error starts decreasing after ~ 2000 years. This seems to be around the same time as the migration begins to halt due to CO_2 trapping. We also see that the error seems to converge quite rapidly to a constant when the migration starts to halt. The maximum error we get after simulating for 5000 years is $\sim 2.22e^{-5}$, and $\sim 7.47e^{-5}$ in the l_2 norm. In the l_1 norm, the error seems to converge to about ~ 0.001, which shows that the error obtained for this problem is small.

If the error introduced here is a problem, there have been proposed solutions for iterative solvers by Göddeke, Strzodka, and Turek [24]. The techniques proposed uses the single precision solution as an initial guess for the double precision solver. It is therefor easy to use this also for an explicit solver with single precision, where you can use an implicit solver for correcting the single precision error. Using implicit solvers are both computationally demanding and a lot harder to implement than our explicit solver, and this approach might increase the simulation time more than desired. Another approach to this is to use a mixed-precision approach, where parts of the simulator, like adding up the contributions from the neighboring faces of a cell, is implemented in double-precision. The speedup for these parts, with respect to the single-precision only approach, will be 1/2 for the Fermi architecture, but only 1/4 for older architectures which support double-precision.



(a) Visual difference in saturation.



(b) Error norms of the difference in height.

Figure 4.6: Comparison of single and double precision. 4.6(a) shows two simulations of the Johansen formation. One using single and one using double precision. 4.6(b) shows the norm of the difference in height of single and double precision.

4.6 Migration pressure

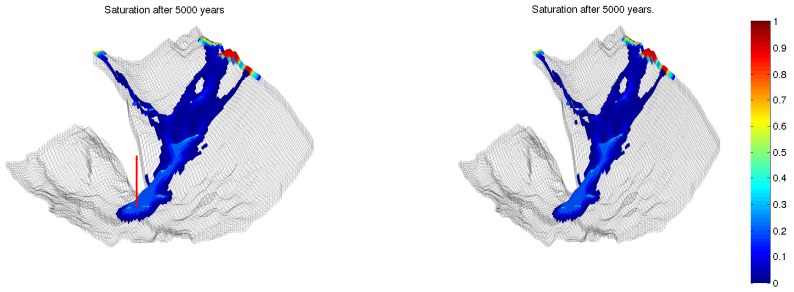
As previously stated, we might be able to do less pressure updates in the migration phase of the simulation, which would decrease the total simulation time as well as reduce the impact of starting threads and copying data to the GPU. This will introduce some errors, and we will start by investigating the errors introduced, before looking at the impact it will have with respect to the simulation times and speedups.

To test the accuracy of changing the number of pressure updates, we have again used the Johansen formation, where we have injected CO_2 for 50 years, and run 4950 years of migration. To test the speedup obtained, we have used both the unstructured model, as well as the FMM model. Both are run with 10 years of injection, and 990 years of migration, where the time is only taken for the migration period. When testing the speedup, we have only updated the pressure at the beginning of the migration period, and only ran transport during the migration period.

4.6.1 Accuracy

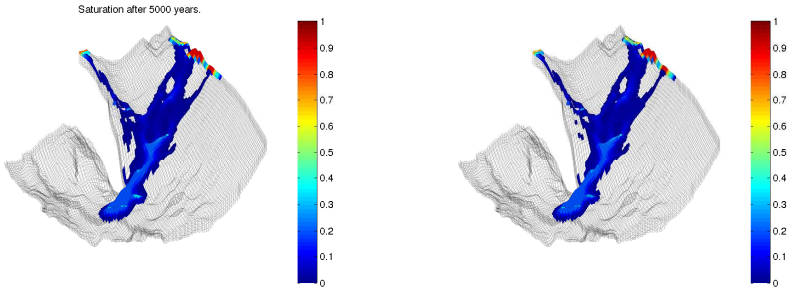
From Figure 4.7 we see that there are no visual difference between updating the pressure every year (4.7(a)) or updating the pressure every 50th year (4.7(b)). We also see that there are some differences between updating every year and updating only once (4.7(c)), especially by the fault where updating once results in flow where there are no flow when updating every year. We also see that there are some errors introduced when exponentially increasing the time step for updating the pressure (4.7(d)), but they are a lot less than the ones introduced by only updating once.

In Figure 4.8 we see that the actual difference between updating the pressure every year, and only every 50th year is very small. The largest error we see is just above $1.0 \cdot 10^{-3}$, and are only present at two places. When updating the pressure only at the start of the migration period, we see that the error is a lot larger. The largest error we have is approximately 0.15, which is a large error. This error is, however, only in one place, and the rest of the errors seems to be less than 0.1, and most of the errors introduced is less than 0.05. This is a lot larger than updating every 50th year, but is quite similar when considering the computational savings. When exponentially increasing the time step for the pressure update, we see that the errors are a lot smaller than when updating the pressure only once, with a maximum error of approximately 0.06. Much of the error is in the range $[0.02, 0.05]$, which must be said to be fairly good, and starts to resemble updating every 50th year. The next aspect of this investigation is to see how this will affect the simulation time, as well as the speedup for the multi-core and GPU solutions.



(a) Pressure update each year

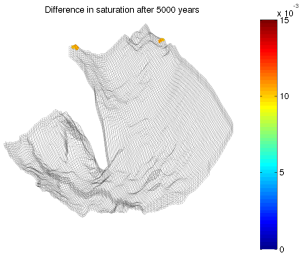
(b) Pressure update each 50th year



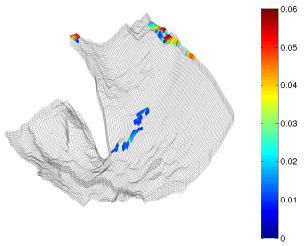
(c) Only one pressure update

(d) Exponential increase in pressure time step

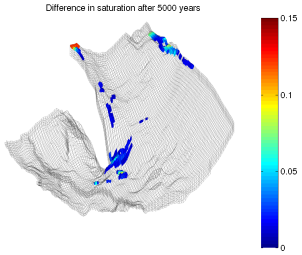
Figure 4.7: Results from simulating with different pressure update steps in the migration period. 4.7(a) is a plot from updating the pressure each year. 4.7(b) is a plot from updating the pressure every 50th year. 4.7(c) is a plot from only updating the pressure at the start of the migration period, and 4.7(d) is a plot from exponentially increasing the time step for the pressure update.



(a) Comparison for updating every 50th year

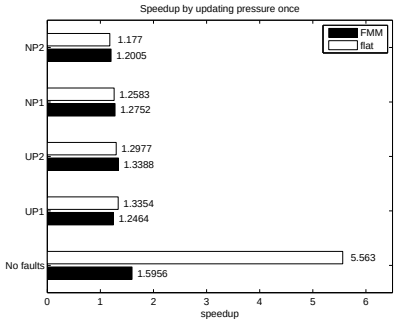


(b) Comparison for exponential updating

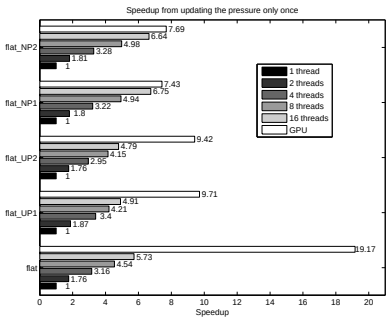


(c) Comparison for updating only once

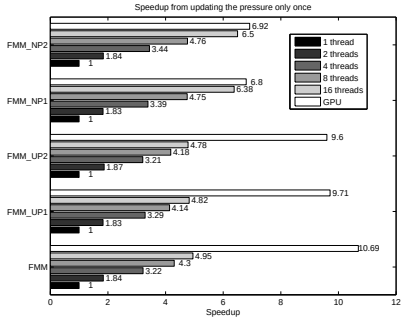
Figure 4.8: The difference between updating the pressure every year, updating every 50th year in 4.8(a), exponentially increasing the time step for the pressure update 4.8(b), and only once in 4.8(c).



(a) Speedup by updating the pressure only once running one thread



(b) Speedup form using the unstructured model



(c) Speedup by using the FMM model

Figure 4.9: Speedup obtained by updating the pressure only once.

4.6.2 Performance

As we can see in Figure 4.9(a), we get a small speedup for one thread by updating the pressure only once for the migration period for most of the models. The speedup is not large, and so we can say that we are close to the limit of our single-threaded solution already when we update the pressure every year. For the unstructured model with no faults however, we see a surprisingly large speedup. The reason for this is that we get a lot looser time step restriction using the unstructured formation with no faults, using 264 steps for the whole migration period and 1698 steps for the FMM formation. This means we have to run one transport step for each pressure update, when running the unstructured formation and updating the pressure every year, while for the FMM formation we have to run four transport steps. Running a pressure update every year, this will not make any noticeable difference because of the overhead of copying data and starting threads.

The speedup obtained by updating the pressure only once is given in Figure 4.9. As we can see, the multi-core solver gains a better speedup in the migration period than it did in the injection period when updating the pressure every year. We also see that the multi-core soler is not much affected by adding faults in the reservoir, and actually handles non-uniform faults better than uniform faults. The GPU on the other hand, seems to handle uniform faults fine, but is affected some by adding non-uniform faults. This might be caused by a divergent branch when vertically integrating the permeability. We also see that the speedups obtained by updating the pressure once in the migration phase, is much better than when updating the pressure every year, but still not as good as in the injection phase of the simulation. We also see a speedup of 19.17 for the unstructured formation with no faults, which further supports what we saw when updating the pressure every year, that the GPU solution is affected by both faults and structures in the top formation.

By comparing the results from Figure 4.9 with the results from Figure 4.5 and Figure 4.4 we see that we have a large overhead by starting threads and copying data to the GPU. This will have a big impact by updating the pressure every year, which can be damped by updating the pressure only once, or more seldom than every year.

4.7 Hardware utilization

One of the criticisms of measuring GPU speedup, is that one have run highly optimized GPU code, which is compared to unoptimized CPU code. It is therefor interesting to look at how both our CPU and GPU solutions utilizes the hardware available to us. We will look at the Flops (floating-point operations per second) we get on the single threaded CPU solver, compared to the same for the GPU.

These tests are run using the FMM formation with the same rock parameters and wells used for the synthetic test suite. The difference is that we, for this test, have injected for 10 years, then updated the pressure once before running

migration for 990 years. This is then profiled, using Intel VTune Amplifier XE Performance Profiler 2011 (VTune) for the CPU solver and the Compute Visual Profiler for the GPU [27, 41]. The tests are done on the GPU testing rig in Table 4.2, since VTune did not run on the other rig due to access restrictions, but the percentages obtained should, in theory, be independent of the CPU.

For the single core CPU solution, we have a throughput of 2.971 GFlops. This is 27.82% of the theoretical flops on one core. The reason we do not use more of the available performance, is that full performance requires using Streaming SIMD Extension (SSE) instructions [26], which makes it possible to do four floating point operations per clock cycle, meaning we work on 128 bits at the same time. For our problem, this would require a complete rewrite of our algorithm and is outside of the scope of our thesis. This means that the theoretical performance on one core, without using SSE instructions are 2.67 GFlops. If we look at the output from profiler however, we see that many of the floating point instructions are compiled into SSE instructions, which is the reason we gain more than 25% of the performance. Many of these instructions however, only work on the last 32 bits of the 128 bit array. We further see that we have 14.92% stalled cycles, meaning we use quite some computational time waiting for data to process.

When calculating the flops using four CPU cores, we have a throughput of 9.368 GFlops, meaning we utilize 21.93% of our CPU. This also corresponds well to the speedup we got when testing updating the pressure only one in Figure 4.9. The reason we do not get a 4 times speedup seems to be that we have to wait for data to compute on, as we have 22.05% stalled cycles when running four cores, which is a large increase with respect to using one core.

For the GPU solution, we see that we have a throughput of 475.11 GFlops through the kernels, meaning we utilize 47% of our GPU. We also have more than 50% stalled cycles, meaning our implementation are waiting quite some time for data to compute on, and is therefor memory bound. This number is expected to decrease by using texture and shared memory, but we also expect the implementation to remain memory bound. This means that we might be able to increase the number of arithmetic operations inside of the kernels, without using more time for the simulations. In other words, we expect a higher order scheme to not impact the simulation time as much for the GPU, as it would the CPU.

Chapter 5

Conclusion

5.1 Summary of Results

Our performance tests shows that the GPU outperforms the CPU both on simple grids as well as on grids with realistic features. We also see that the GPU scales better than the CPU when changing the resolution of the grid in the horizontal direction, but the GPU is more affected than the CPU when refining the vertical direction.

We see that this problem utilizes the GPU hardware better than the CPU, but that the implementation is memory bound, meaning we can not utilize the full theoretical performance of the GPU. It also shows that, for parallel problems, it is easy to achieve a high throughput on the GPU.

While testing the accuracy, we saw that using single-precision floating-point numbers on the GPU introduces a small error each time step, but that the slope of the error decreases as the injection period ends. We have also shown that the error introduced is not visible in the visualized result, and discussed why the error is negligible.

Further we have looked on ways to speed up the migration period by only updating the pressure at the start of the migration period, or increase the time step for the pressure. We have shown that the errors introduced might not be possible to neglect, but shown that they will give an indication to how the CO_2 will flow in the reservoir. We have also shown that these changes will decrease the simulation time considerably.

One possible problem is the memory size on the GPU. The 896MB of memory on the GPU will clearly limit the size of the grids we can use. This will be a problem as the resolution of the grids increases.

5.1.1 Performance benefits

From our GPU and CPU solver we see that the GPU scales a lot better than the CPU when simulating our problem. This coincides with work done on other parallel problems. We also saw that the CPU handles increase in vertical resolution better than the GPU solver, which is probably caused by possible divergent branches and uncoalesced memory reads, when integrating the permeability.

The GPU does not only give us more flops, meaning better theoretical performance, but also more flops per watt, keeping the GPU at a constant power consumption, but increasing the performance by adding more cores is still possible, and one can expect a further increase in power efficiency in the future. The GPU also gives us more flops per dollar, which makes it even more interesting from a consumer perspective.

5.1.2 Accuracy

Possibly the strongest argument against using the GPU as an accelerator unit, is the usage of single precision. However, we have shown that the error introduced by using single precision, for this particular problem, is negligible with respect to using double precision. Later generations of GPUs also have better double precision performance than the GPU used in these tests, and the performance difference between single and double precision is now the same on the newest generation of Nvidia GPUs, as it is on current CPUs. We have also proposed methods to correct errors introduced using single precision, by using the single precision solution as an initial guess as well as a mixed precision solver. These methods have previously been proven to give good results.

Increasing the time step for the pressure update during the migration period has been shown to be beneficial with respect to simulation time, as well only introducing a small error. This shows than one should increase the time step significantly during the migration period and one can actually do only one pressure update at the start of the migration period to give an indication of how the CO_2 will flow. For a thorough risk analysis one should investigate how large time step one can use for the migration period, by starting with a large time step and decrease it until there is almost no change in the result. As the pressure driven velocity will decrease also during the migration period, one can also start with a small time step for the pressure update in the migration period, and increase the time step during the migration. This will also make the total simulation a lot faster, as the pressure update routine is currently a lot slower than the transport implementation.

5.2 Further work

There are still many aspects of CO_2 migration that need to be tested on the GPU, as well as modifications to the GPU code that might give it an even better speedup. Implementing the use of texture and shared memory, as well as implementing the time step calculations on the GPU will be the first step,

but possibly more interesting is the implementation of higher-order solvers and support for multiple GPUs.

5.2.1 Multiple GPUs

Using multiple GPUs would give us a lot more theoretical performance, but the implementation of support for multiple GPUs is not a trivial task. A splitting of the data between the GPUs is necessary, and after a new pressure flux is calculated one would need to split that flux appropriately, to make sure the correct fluxes are sent to the correct GPUs. Further, one would need to assemble the contributions from the different GPUs again when the simulation is done. Along with the increased performance, this would also help with the memory issues, as the requirements are split between several GPUs.

5.2.2 Higher-order solvers

Other work have shown that the computation time does not increase as much on the GPU when introducing solvers of higher order. The reason for this is that an increase in arithmetic operations does not affect the GPU as much as it does the CPU, because first-order solvers are mostly memory bound, while introducing higher-order solvers would ultimately make it computation bound instead [10]. To test how this problem scales on the GPU when introducing higher-order schemes would be interesting, and the GPU would most likely not be affected to the same extent as the CPU solver would.

5.2.3 Adaptivity

Right now, one can either run a threaded CPU solver, or a GPU solver. Our tests shows that the GPU solver will be the fastest solver, but can not handle too fine grids. One could extend the program to adaptively choose the solver. This could be done by checking the memory available on the device at runtime, as well as check the size of the data. Then the program could choose to send it to the GPU if there is enough memory, or choose the appropriate number of threads to use to solve it on the CPU. This would require a query of the number of cores on the CPU. This is easy to do in OpenMP, which is the library used for multi threading in this thesis.

Bibliography

[1] J. E. Aarnes, T. Gimse, and K.-A. Lie. "An introduction to the numerics of flow in porous media using Matlab". In: *Geometrical Modeling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF*. Springer Verlag, 2007, 265–306.

[2] I. Akervoll and P. E. S Bergmo. "A study of Johansen formation located offshore Mongstad as a candidate for permanent CO_2 storage". *ccs conference*. 2009.

[3] R. Allam et al. "Capture of CO_2". In: *Carbon Dioxide Capture and Storage*. Cambridge University Press, 2005, 105–178.

[4] J. Anderson et al. "Underground geological storage". In: *Carbon Dioxide Capture and Storage*. Cambridge University Press, 2005, 195–276.

[5] J.R. Appleyard et al. "Accelerating Reservoir Simulators using GPU technology". *SPE Reservoir Simulation Symphosium*. 2011.

[6] M. de la Asunciòn, J.M. Mantas, and M.J. Castro. "Simulation of one-layer shallow water systems on multicore and CUDA architectures". *Journal of Supercomputing* (2010). [In press].

[7] J. Backus. "Can programming be liberated from the von Neumann style?: A functional style and its algebra programs". *Commun. ACM* 21.8 (1978), 613–641.

[8] A. Baklid, R. Korbol, and G. Owren. "Sleipner vest CO_2 Disposal, CO_2 injection into a shallow underground aquifer". *SPE Reservoir Simulation Symphosium*. 1996.

[9] P. Brewer et al. "Ocean storage". In: *Carbon Dioxide Capture and Storage*. Cambridge University Press, 2005, 277–318.

[10] A.R. Brodtkorb et al. "Simulation and visualization of the Saint-Venant system using GPUs". *Computing and Visualization in Science* 13.7 (2010), pp. 341–352.

[11] A.R. Brodtkorb et al. "State-of-the-Art in Heterogeneous Computing". *Scientific Programming* (2010), pp. 1–33. ISSN: 1058-9244.

[12] A. Bydal. "GPU-accelerated simulation of flow through a porous medium". 2009.

[13] Petroleum Technology Research Center. *Weyburn-Midale CO_2 project*. [Visited 2011-03-08]. 2011. URL: `http://www.ptrc.ca/weyburn_overview.php`.

[14] H. Class, A. Ebigbo, et al. "A benchmark study on problems related to CO2 storage in geologic formations". *Computational Geosciences* 13.4 (2009), pp. 409–434.

[15] CO_2 Capture Project. *CO_2 Trapping Mechanisms*. [Visited 2011-03-08]. 2000. URL: `http://www.co2captureproject.org/co2_trapping.html`.

[16] K. H. Coats et al. "The Use of Vertical Equilibrium in Two-Dimensional Simulation of Three-Dimensional Reservoir Performance". *SPE Journal* 11 (1971).

[17] D. Coleman et al. "Transport of CO_2". In: *Carbon Dioxide Capture and Storage*. Cambridge University Press, 2005, 179–194.

[18] H.K. Dahle et al. *A model-oriented benchmark problem for CO_2 storage*. [Visited 2011-04-19]. 2009. URL: `http://arks.princeton.edu/ark:/88435/dsp01pn89d657g`.

[19] H. Darcy. *The public fountains of the city of Dijon*. Trans. by P. Bobeck. Kendal Hunt Publishing, 2004.

[20] J. Dupuit. "Études Théoriques et Pratiques sur le Mouvement des Eaux dans les Canaux Decouverts et à Travers les Terrains Perméables". *Dunod* (1863).

[21] G. Eigstad et al. "Geological modeling and simulation of CO_2 injection in the johansen formation". *Computational Geosciences* 13.4 (2008), pp. 435–450.

[22] S.E. Gasda, M.A. Celia, and J.M. Nordbotten. "Upslope Plume Migration and Implications for Geological CO2 Sequestration in Deep, Saline Aquifers". *IES Journal A: Civil and Structural Engineering* 1 (2008), pp. 2–16.

[23] S.E. Gasda, J.M. Nordbotten, and M.A. Celia. "Vertical equilibrium with sub-scale analytical methods for geological CO_2 sequestration". *Computational Geosciences* 13.4 (2009), pp. 469–481.

[24] D. Göddeke, R. Strzodka, and S. Turek. "Accelerating double precision FEM simulations with GPUs". *18th Symposium Simulationstechnique*. Frontiers in Simulation. SCS Publishing House e.V., 2005, 139–144.

[25] Intel. *Inside Intel Nehalem Microarchitecture*. [Visited 2011-02-22]. 2009. URL: `http://www.notur.no/notur2009/files/semin.pdf`.

[26] Intel. *Intel(R) 64 and IA-32 Architectures: Software Developer's Manual*. 2011.

[27] Intel. *Intel(R) VTune(TM) Amplifier XE 2011 Getting Started Tutorials for Linux* OS*. Version 1.0. 2011.

[28] Intel. *Sh Embedded Metaprogramming Language*. [Visited 2011-03-20]. 2010. URL: `http://libsh.org/index.html`.

[29] W. Kahan. *A Logarithm Too Clever by Half*. [Visited 2011-04-20. 2004. URL: `http://www.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT`.

[30] Inc. Khronos Group. *The OpenCL Specification*. Version 1.0. 2009.

[31] H. Klie et al. "Exploiting Capabilities of Many Core Platforms in Reservoir Simulation". *SPE Reservoir Simulation Symphosium*. 2011.

[32] D.E. Knuth. "Structured Programming with go to Statements". *Computing Surveys* 6.4 (1974), pp. 261–301.

[33] R.J. LeVeque. "Finite Volume Methods for Hyperbolic Problems". In: Cambridge University Press, 2007.

[34] I. Ligaarden and H. M. Nilsen. "Numerical aspects of using vertical equilibrium models for simulating CO2 sequestration". *12th European Conference on the Mathematics of Oil Recovery*. 2010.

[35] C.W. MacMinn and R. Juanes. "Post-injection spreading and trapping of CO2 in saline aquifers: impact of the plume shape at the end of injection". *Computational Geosciences* 13.4 (2009), pp. 483–491.

[36] J.C. Martin. "Partial Integration of Equations of Multiphase Flow". *SPE Journal* 8 (1968), pp. 370–380.

[37] J.C. Martin. "Some mathematical aspects of two phase flow with application to flooding and gravity segregation". *Prod. Monthly* 22 (1958), pp. 22–35.

[38] SINTEF Applied Mathematics. *MRST - MATLAB Reservoir Simulation Toolbox*. [Visited 2011-03-25]. 2009. URL: `http://www.sintef.no/mrst`.

[39] MatMoRA. *Geological Storage of CO_2: Mathematical modelling and Risk Analysis*. [Visited 2011-04-24]. 2007. URL: `http://www.sintef.no/Projectweb/MatMoRA`.

[40] G. Moore. "Cramming more components onto integrated circuits". *Electronics* 38.8 (1965), pp. 114–117.

[41] Nvidia. *Compute Visual Profiler User Guide*. Version 2. 2010.

[42] Nvidia. *CUDA community showcase*. [Visited 2011-02-22]. 2011. URL: `http://www.nvidia.com/object/cuda_showcase_html.html`.

[43] Nvidia. *Nvidia CUDA C Best Practices Guide*. Version 3.2. 2010.

[44] Nvidia. *Nvidia CUDA C Programming Guide*. Version 3.2. 2010.

[45] R.M. Ramanathan. *Intel Multi-Core Processors: Making the Move to Quad-Core and Beyond*. Tech. rep. Intel, 2006.

[46] F. Riddiford, S. Akretche, and A. Tourqui. "Storage and Sequestration of CO_2 in the In Salah Project". *World Gas Conference, Tokyo*. 2003.

[47] E. Rubin et al. *Carbon Dioxide Capture and Storage*. Cambridge University Press, 2005.

[48] G. Sand and A. Braathen. "CO_2-fritt Svalbard i 2025?" *Svalbardposten* (2006).

[49] Statoil. *Carbon storage started on Snøhvit*. [Visited 2011-03-08]. 2008. URL: `http://www.statoil.com/en/NewsAndMedia/News/2008/Pages/CarbonStorageStartedOnSn\%C3\%B8hvit.aspx`.

[50] Univeristy of Svalbard. *The Longyearbyen CO_2 Lab*. [Visited 2011-01-07]. 2007. URL: `http://co2-ccs.unis.no/home.html`.

[51] Standford University. *BrookGPU*. [Visited 2011-03-20]. 2007. URL: `http://graphics.stanford.edu/projects/brookgpu/`.